AD-A046 312 STANFORD RESEARCH INST MENLO PARK CALIF F/G 12/2
THE SCHEDULERS OF ACS.1.(U)
SEP 77 M C PEASE N00014-77-C-0308
UNCLASSIFIED SRI-TR-14 NL

1 OF 2
AD A046 312

Technical Report 14
September 1977

THE SCHEDULERS OF ACS.1

By: MARSHALL C. PEASE

Prepared for:

OFFICE OF NAVAL RESEARCH
DEPARTMENT OF THE NAVY
ARLINGTON, VIRGINIA 22217

Contract Monitor: MARVIN DENICOFF, PROGRAM DIRECTOR
INFORMATION SYSTEMS BRANCH

CONTRACT N00014-77-C-0308

SRI Project 6289

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

(9) Technical Report 14
(11) September 1977

(12) 114 p.

(14) SRI-TR-14

(6) # THE SCHEDULERS OF ACS.1

(10) By: MARSHALL C. PEASE

Prepared for:

OFFICE OF NAVAL RESEARCH
DEPARTMENT OF THE NAVY
ARLINGTON, VIRGINIA 22217

Contract Monitor: MARVIN DENICOFF, PROGRAM DIRECTOR
INFORMATION SYSTEMS BRANCH

(15) CONTRACT N00014-77-C-0308

SRI Project 6289

Approved by:

JACK GOLDBERG, *Director*
Computer Science Laboratory

EARLE D. JONES, *Executive Director*
Information Science and Engineering Division

ABSTRACT

ACS.1, for "Automated Command Support," is a research system for studying uses of knowledge-based systems for the support of management. The research addresses the managerial responsibilities for planning operations, maintaining and executing approved plans, and for the retrospective analysis of the results of operations. The purpose of the research is to develop architectural principles for the design of intelligent management support systems.

Viewed from the top level, ACS.1 can be regarded as an assembly of modules called "schedulers" and "planners," with certain other modules and subsystems which are not of direct concern here. The planners "know how to" plan certain types of operations. The schedulers "know how to" coordinate the expected use of particular types of resources, whether human, equipment, supplies, or facilities. The planners have responsibility for creating detailed plans to meet specified objectives, including the timing of all required tasks and determining the assignment of all necessary resources. The schedulers are responsible for assigning the resources so as to avoid conflicts with other plans or expected events.

This report describes the design of the schedulers. It details the functions that implement the operations required of a scheduler, as well as discussing the reasons for the implementations chosen. In developing these implementations, a number of technical problems have been addressed and solutions developed. The technical features developed to meet these problems include the following:

The scroll table as a convenient medium for interaction with the system's other elements and with the manager.

Resource models to define the knowledge used by the schedulers, and structures to encode them making them available to the manager for modification or extension. The identification of the resource model as an explicit structure makes the model available to the manager so that he can adapt the scheduler to meet new requirements and situations.

The specific forms that have been developed to implement various features. These include scroll tables for orga[illegible], the data, means used to encode the resource models, a[illegible] structures to enforce self-consistency and to obtain wha[illegible] call the "self-" and "mutual-destruct" properties.

The techniques used to realize these features have been chosen to provide considerable generality. We believe that they are applicable to a wide variety of systems that are intended for the support of management.

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ACKNOWLEDGMENTS

The work described here has been carried on under the direction of Jack Goldberg. Richard Fikes has had a very important part in the development of the concepts and approach. Steven Weyl has also contributed to the early work, and Kazutaka Tachibana did much of the early programming. Daniel Sagalowicz currently is participating in the program, and has contributed significantly to its present state.

I. INTRODUCTION

This report describes the modules called schedulers in the experimental system, ACS.1 (for Automated Command Support). This system is intended as a vehicle for the development of techniques for building knowledge-based systems that will provide intelligent support to a manager. The areas of support addressed are planning of operations, administration and monitoring of approved plans, and retrospective analysis of those operations. Viewed from the top level, the system as a whole appears as a system of autonomous modules, some of which are called "schedulers" and have the responsibility of coordinating the use of specified types of resources. The other modules, called "planners," have the responsibility for planning specified types of activities. This report addresses the design of schedulers, and describes other components only to the extent necessary to understand the requirements of the schedulers. The schedulers are of special interest since the design principles and techniques used in their implementation may be useful for other applications.

ACS.1 has operated in the simulated environment of a naval air squadron, although the techniques used lend themselves to a wide variety of application environments. The principle operations being planned and managed are flight missions. This requires coordinating such various resources as pilots, aircraft, maintenance crews, deck crews, launch facility and crew, and recovery facliities and personnel. There are also additional demands on these resources such as the pilots' need for rest, or equipment maintenance. Other events can limit the availability of certain resources, such as a pilot becoming sick or an aircraft requiring unexpected maintenance. The scheduler's function is this coordination, maintaining the information necessary to achieve it. The challenge of the scheduler design is the result of the variety of possible situations, the complexity of possible interactions among the demands for a given type of resource, and the possibility that future availability of a resource can change in important ways at any time.

The system concept, and its development in terms of the desired application, has been described in some detail in Technical Report 13, "ACS.1: An Experimental Management Tool," (1977). In that report, the relation of this work to other research in artificial intelligence and in optimal scheduling is discussed. That material will not be repeated in this report, although a brief overview of the system concept is given to place the requirements for the schedulers in context.

After a brief overview of the system's requirements and concept, detailed requiremnts for a scheduler are discussed. Four aspects of scheduler operations are considered in some detail: the data structure used by a scheduler, the way its knowledge is encoded, the precise definition of what is required to maintain self-consistency in its data, and the requirement for alert functions and other capabilities for initiating system action. Following this, the functions used by the schedulers are given and described in detail.

II. SYSTEM CONCEPT

The main elements of the system (shown in Figure 1 in block diagram) are modules called "planners" and "schedulers," each of which is responsible for a well-defined part of the system's operation. A planner develops plans for a specific type of activity. A scheduler coordinates the planned usage of a specific type of resource. Interactions among the modules are processed entirely through messages passed through the "message handler" unit. All communications to or from a user, or to or from the data system also pass through the message handler. The use of the message handler is an important feature that helps maintain the autonomy of the separate modules and provides a central switching location for user control of the system's operation.

Between the user's terminal and the message handler is the user interface, which provides a pseudo-natural language capability. It uses a language facility called LIFER, developed by the Artificial Intelligence Center at SRI International for other purposes. The user interface accepts requests or commands in a natural language format. It is not a true natural language capability; it uses pattern recognition rather than syntactic and semantic analysis. The user is required to use one of a predetermined set of commands or questions, which can be extended dynamically. The package includes an automatic facility for handling elliptic inputs, for correcting spelling or grammar, and a mechanism for allowing even novices to extend the language recognized by the system through paraphrase. It provides a convenient interface facility.

The data system has not been implemented yet. Plans call for it to be more than a simple repository of data. It will have the responsibility for monitoring the execution of approved plans, checking that tasks are started and completed as planned, and that the resources needed for an operation are available as required. It will have the responsibility for recognizing when replanning may be necessary, and can initiate replanning as needed. So far, this monitoring function has been handled by a separate module not shown in Figure 1.

The knowledge contained and used by a planner describes a particular type of process to a given level of detail. It identifies the tasks that must be completed during its execution and the required partial ordering among these tasks. It identifies what resource types must be assigned, and their relations to the tasks. It also identifies what other planners must be called to develop the details of any tasks that must be further decomposed, and the schedulers that are responsible for the necessary assignments. It has the information necessary to permit it to initiate requests for subplans and assignments.

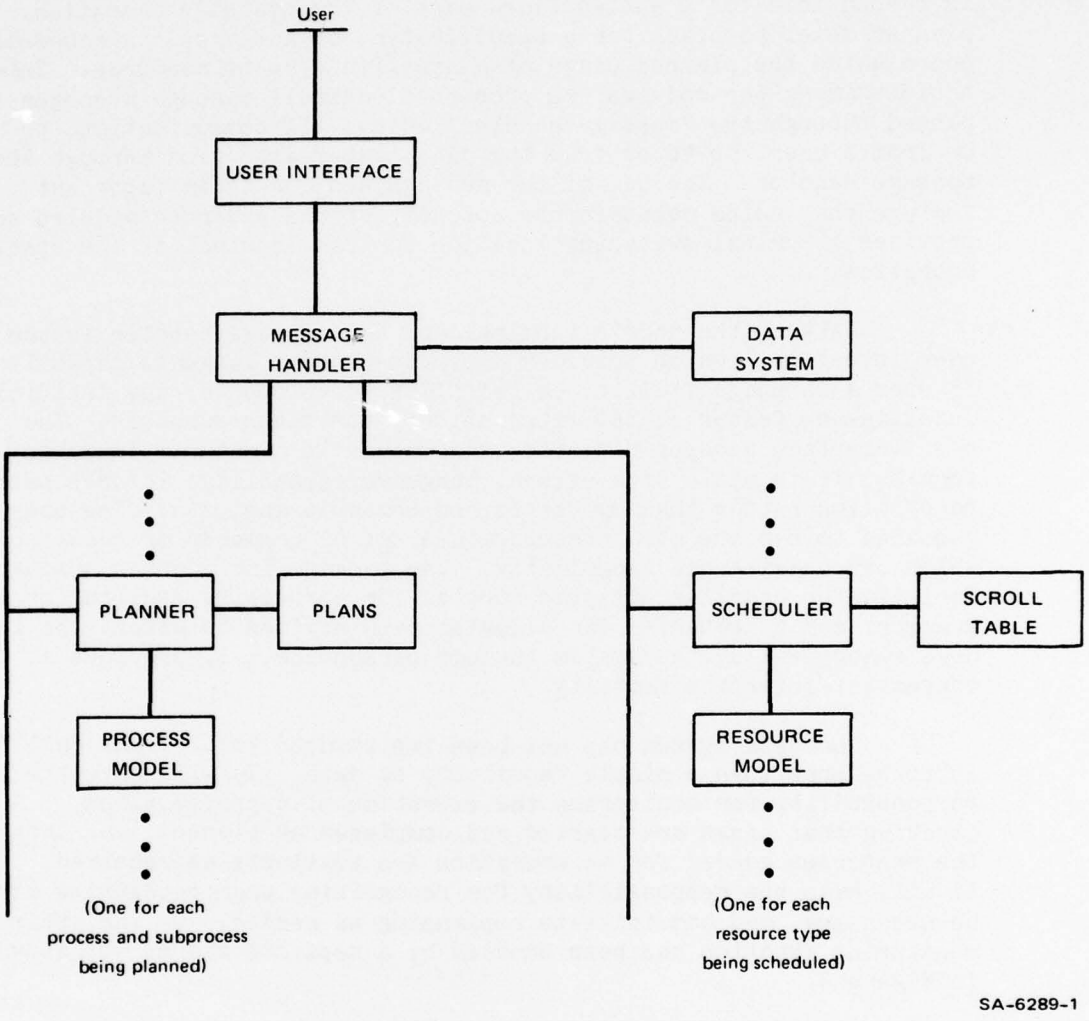

FIGURE 1 BLOCK DIAGRAM OF ACS.I

A task recognized by a planner as a component in its process may require planning itself. If so, some other planner has the responsibility for planning it, perhaps decomposing it further into subtasks and obtaining further assignments of resources. In response to a particular requirement, the system of planners may be structured into a hierarchy of modules operating at various levels of detail. Different hierarchies may be required in response to different requirements, and established through the knowledge contained in the various planners and implemented through the message handler.

Each scheduler has the responsibility for a particular type of resource, whether human, equipment or facilities. In response to a request for assignment of one of the resources of its type for some future interval of time, the scheduler first determines which of its resources will be available for the specified period. If more than one, and if it has been given the authority, it selects one according to criteria within its knowledge and makes the assignment. If it has not been given the authority, it sends the relevent information to the human user for his decision. If no resource is available, the scheduler may, depending on its knowledge of what is required, return the assignment that comes closest to matching the requirement, or refuse the request.

As an example, consider the application environment that has been studied-- the command of a naval air squadron. A flight mission is one activity that needs planning. The commander may enter a requirement that a given mission be planned to reach a specified target at a specified future time, and to leave that target at some later time. This requirement is transmitted to the planner that knows about planning a mission. The process model used by that planner decomposes the activity of flying a mission as shown in Figure 2. The tasks it recognizes are preflight preparation of the aircraft, briefing the pilot, the flight out, the action of the flight itself, the postflight service of the aircraft, and the pilot debriefing. It also knows that a pilot and an aircraft must be assigned.

The preflight preparation of the aircraft may be decomposed further by another planner into the transfer of the aircraft to the flight deck, its preflight service, arming, fueling, and its transfer to he launch facilities. Additional resources, such as maintenance personnel, may be required during some of these subtasks.

A plan has been generated, and will be returned to the commander for his approval or modification, only when all tasks and subtasks have been planned and when all resources needed during any task or subtask have been assigned.

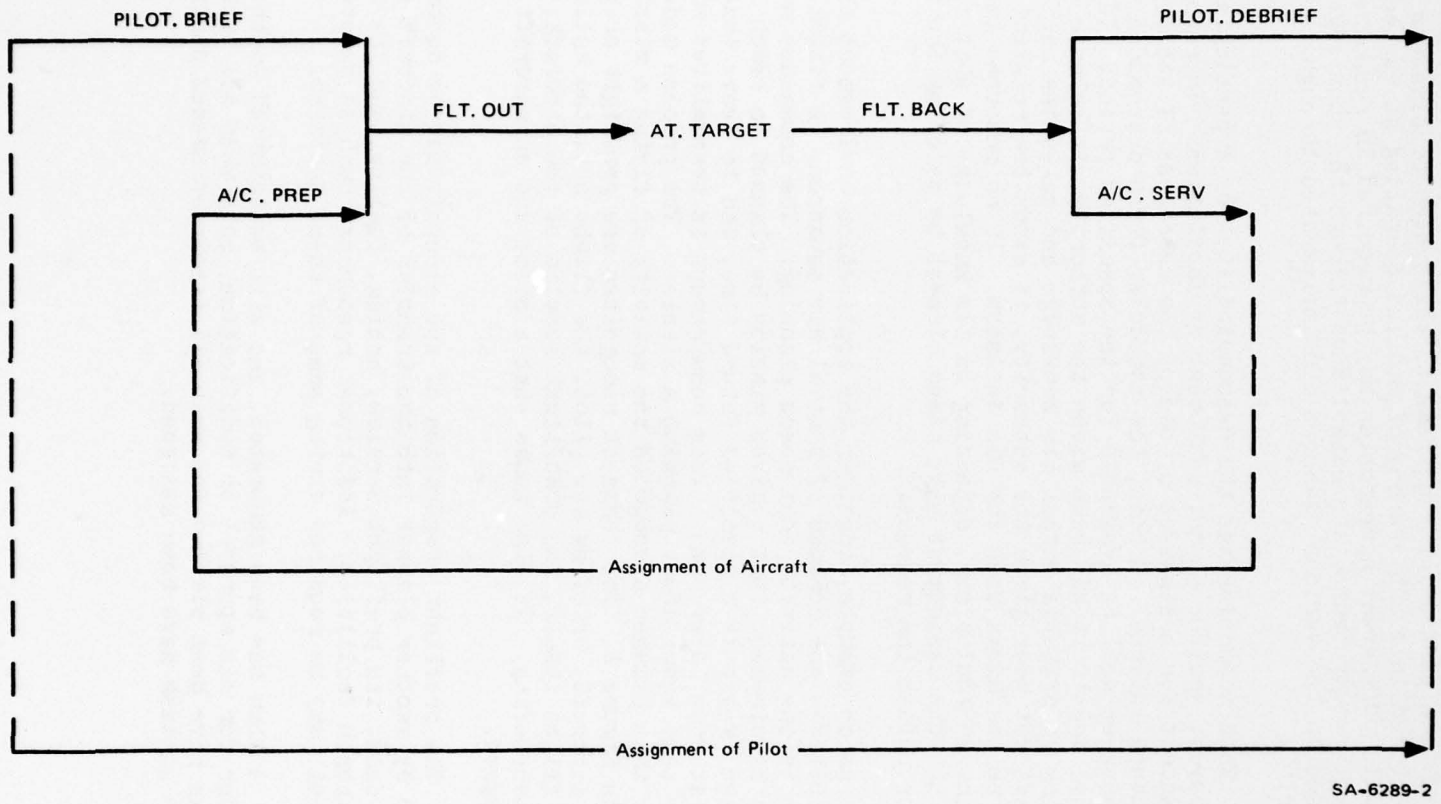

FIGURE 2 PROCESS MODEL FOR FLYING A MISSION

In making assignments, note that a scheduler must be cognizant of all expected future usage of the resources for which it is responsible. For example, the aircraft scheduler must know which aircraft have been assigned to other missions or are due for scheduled maintenance actions or otherwise unavailable. It must respond to a request for assignment in a way that is consistent with that expected future usage. The specification of what is meant by consistency for a particular type of resource is contained in what the resource model of the scheduler.

There are several features of the system concept that are of prime importance to the type of application being considered. These features have dominated the development of the experimental system, and have been an important component of the research. They can be summarized as follows:

The division of responsibilities among the planners and schedulers should correspond to the division of responsibility in the comparable human organization. This feature helps the user to understand the system's operations, and should permit the orderly growth of the system. It also facilitates the transfer of responsibility between the system and the human organization to handle exceptional situations.

The knowledge contained by the planners and schedulers should be explicit and accessible for modification without major revision of the system. This is considered necessary to permit adapting the system to changing needs and conditions. It also permits introducing new planners or schedulers through specification of the applicable knowledge. This feature permits the rapid extension of the scope of the system, or its transfer to new application environments.

The scope and operation of each planner or scheduler should be sufficiently simple to make it readily understandable by the human user. The complexity of system operation should be the result of interactions among the modules, rather than contained within any module. Again, this facilitates growth and adaptation, and permits rapid modification to meet exceptional conditions.

Further details of the system concept, and of the techniques that have been used to implement it, have been given in Technical Report 13, and are not repeated here. In the next section, we consider the requirements for the schedulers in greater detail.

III. SCHEDULER REQUIREMENTS

There are a number of requirements schedulers should meet. These derive from consideration of their roles in the planning and monitoring functions of the system, and as facilities that maintain information about the future use of their resources. In particular, the schedulers are required to:

Respond to planning needs.

Assist in maintaining plans by recognizing conflicts.

Support the user's need for overviews of resouce commitments.

Support the need for alert function.

Initiate system-originated planning as required.

In addition, the schedulers should be designed to support the need for adaptation.

A scheduler's primary system function is responding to requests for assignment of its resources, or determining that no assignment is possible. For this purpose, it must maintain all relevent all information affecting the availability of its resources. It must also have the knowledge and procedures that can act on this data to produce an appropriate response to a request for assignment.

A scheduler also is required to recognize when new data invalidates previous assignments. New information about the expected future state of a resource can be entered at any time. It is the scheduler's responsibility to determine if this data creates a conflict, and, when it does, to initiate the appropriate action.

A scheduler contains detailed information about the expected future use of its resources. This information can be important to the manager, and should be available to him in a format convenient to his needs.

The scheduler can provide some of the alert functions that may be required of the system. The term, alert function, means one that watches for the occurence of some condition, and, when it occurs, issues a message advising the manager of the fact. For example, the manager may want to be warned when there is danger of overload. The scheduler can be directed to monitor the loading of its resources, and to issue an alert message when a specified level is exceeded.

The scheduler also can be required to initiate other system actions. For example, in the application environment of the naval air squadron, the aircraft scheduler can be directed to keep track of the accumulated flight hours for each aircraft and to initiate the planning of scheduled maintenance when required.

In addition, there is the general requirement, discussed above in Section II, of making a scheduler's knowledge accessible for modification and adaptation. It is considered vital that a user who may not have expert knowledge of the system be able to modify the knowledge it contains, or to create new schedulers as needed to handle resource types not previously included in the system.

The following sections describe the techniques that have been developed to meet these requirements.

IV. SCHEDULER OPERATIONS

There are four significant aspects of scheduler operations. These are considered separately in the following subsections which discuss the following topics:

Data Structure-- The way a scheduler holds information about expected events and plans.

Specification of the Model-- The means for encoding knowledge in a scheduler, and for modifying it as required to adapt a scheduler to new requirements.

Self-Consistency in a Scheduler-- The means for enforcing continued self-consistency of data contained in a scheduler as defined by its resource model.

Alert and System-Initiated Actions-- The techniques used to set and modify alert functions and other processes that can initiate planning or other system actions.

In the following subsections, these aspects are discussed from the viewpoint of the system user, describing the system behavior he can observe and have available for use. The functions that implement this behavior are discussed in the following section.

A. Data Structure

The data structure used by a scheduler is what we call a "scroll table." Conceptually, a scroll table is a two-dimensional array. Each row represents a named one of the resources being handled by the scheduler. Each column represents an interval of time specified for the scheduler. The first column always includes the current value of the simulation time used by the system. As current time advances into the second column, the first column is dropped and the second column becomes the first. The table is said to have been scrolled. In the table, data is held in the "cells" of the applicable row of each column.

It should be emphasized that this is a conceptual description of a scroll table. Its actual implementation is somewhat different, as is described shortly. However, it can be printed in the form described.

To illustrate, Table 1 shows the scroll table for the pilots of ACS.1 as it might exist following certain assignments and the entry of certain data.

Table 1
PILOT SCROLL TABLE

<MONITOR>: SHOW THE TABLE FOR THE PILOTS FROM 0:00:00 TO 0:03:00

Table: PILOTS
Property: STATE-NAME (AVAIL replaced by *.)

Name\Time	0:00	0:30	1:00	1:30	2:00	2:30
ABLE	SICK	SICK	SICK	SICK	SICK	SICK
BAKER	AVAIL	ASG	ASG	ASG	ASG	ASG
CHARLES	*	SICK	SICK	SICK	SICK	SICK
DAVIS	*	ASG	ASG	ASG	ASG	ASG
ELLIS	*	*	*	*	*	*

The entries in Table 1 are the state names in each cell. The state name is a code word that identifies the type of entry that has been made in the given cell. It is one of a set of state names that have been defined by the resource model of the scheduler. In Table 1, ASG means "assigned to an approved plan for a mission". ASG.RET indicates a rest period following the completion of a mission. SICK and AWAY indicate the entry of data identifying that the given pilot is sick or on leave, respectively. AVAIL is the default state, standing for available. The printout of the state names provides a convenient summary view of the expected use of the resources. Much more information is actually contained in each cell, as indicated later.

In Table 1, the first line, after <MONITOR>:, is the actual input command recognized by the pseudo-natural language interface and interpreted into the appropriate function call. The columns represent intervals of time, which, in this case, are each 30 minutes long. The length of this interval is specified at the time the table is set up, and remains a parameter of the scheduler. The rows are labeled by the pilots' names given to the scheduler. The number of rows available in the scheduler, whether used or not, is specified also at the time the table is created, but may be changed later if necessary. Only the rows used are printed in Table 1. In this case, the number of available rows is ten, but only five are being used.

As implemented, the scroll table, as shown in Table 1, exists only as a virtual entity. The principle difference is that columns are created only as needed. This is done to permit the scroll table to include data that may be indefinitely in the future without having to assign an indefinite amount of storage. (To be precise, the limit

on the time is the value, in minutes, which is the largest number recognized by the computer as an integer. In the present system, this time is (2^35 - 1) which, although finite, is large enough for all practical purposes.) Table 2 shows a printout of the same data as that shown in Table 1, but in condensed form in which only the columns that have been generated and that may contain new information are printed.

Table 2
CONDENSED PILOT SCROLL TABLE

<MONITOR>: SHOW THE CONDENSED TABLE FOR PILOTS

Table: PILOTS
Property: STATE-NAME (AVAIL replaced by *.)

Name\Time	00	00:30	03:30	09:30
ABLE	SICK	SICK	SICK	SICK
BAKER	*	ASG	ASG.RET	*
CHARLES	*	SICK	*	*
DAVIS	*	ASG	ASG.RET	*
ELLIS	*	*	*	*

The second column of Table 2 covers the period from 0:30 through 2:00 because there is no change in the data between these limits. Similarly, the third column covers the period from 2:00 to 5:00. The final column is from 5:00 through the indefinite future.

In the situation described by Table 2, if data is to be entered for, say, Ellis from 1:30 to 7:30, the first action is to create columns that start at 1:30 and 7:30. The column starting at 1:30 is given the same data as the column starting at 0:00; that start ing at 7:30 is a duplicate of that starting at 5:00. Once these columns have been made, the data for Ellis can be entered.

It should be added that no check is made on whether a column, once made, continues to be needed. For example, if the data for Ellis from 1:30 to 7:30 is later removed, these columns are no longer needed. They are not removed but are left to be eliminated as current time advances and scrolling occurs.

The validity of this procedure for dealing with new and obsolete columns depends on the assumption that most of the data different from the default state is likely to occur in the near future. This assumption is reasonable for the type of application considered. As long as this asumption is true, the average rate at which columns are removed through scrolling can balance the average

rate at which columns are generated, without requiring an excessive number of columns. On the average, the amount of memory used for each scroll table is not excessive.

In its implementation, no data is actually contained in the cells of the table; instead, the cell contains a pointer to a separate data structure. Consequently, an unlimited amount of data can be referenced in each cell, and memory does not need to be assigned unnecessarily.

Additional information, besides the state name, is entered into the cells of a scroll table. Typically, this includes the start and end times of the entry of which the cell holds a part. It also may include an identification code of either that entry or of an associated one as is illustrated shortly. There is a provision, also, for entering other information that may describe, for example, the purpose of the entry. This last part of the entry could even be a textual description of background information that might be important to the manager. We have not used such textual material, but the capability for handling it is provided.

To illustrate, Table 3 shows a possible query addressed to the pilot scheduler that was addressed in Tables 1 and 2.

Table 3
PILOT ASSIGNMENTS

<MONITOR>: WHAT IS THE ASSIGNMENT OF EACH PILOT AT 0:02:15

Name	State	Start	End	ID	FPC
ABLE	SICK	0	INDEF		
BAKER	ASG.RET	120	285	M1	A4
CHARLES	AVAIL				
DAVIS	ASG.RET	120	290	M2	A7
ELLIS	AVAIL				

This printout shows not only the state names for each pilot at the indicated time, but gives a considerable amount of other information as well. For Able, it shows that there is no information about when he is expected to return from sick leave. For Baker and Davis, both are in the rest period following an assignment, as indicated by the state entry. It also gives the start and end time of that state and the identification code and the FPC (flight purpose code) for the mission that caused the state ASG.RET. Note that the start of ASG.RET is the start time of the first column that contains the state. Its end time is the actual one of the mission plus the

rest period set by policy, here assumed to be three hours. The inclusion of the mission's ID allows recovery of the details of the mission itself. The data shown is that obtained by accessing the virtual contents of the corresponding cells of the scroll table that include the specified time, 2:15.

Other data structures can also be attached to the cells. In particular, the demons that enforce the continued self-consistency of the data in the table, to be described in detail later, are data structures that are made part of the content of the cells to which they apply.

Although not shown in the table printouts, there is also the possibility of attaching data to the rows, columns, and the table as a whole. For example, data attached to the rows can be used to accumulate the flight hours during the month for that pilot. The accumulated flight hours for the squadron can be recorded in a data structure attached to the table as a whole. Data about the environment, such as the expected weather conditions, can be attached to the columns.

It is convenient, also, to attach data to the table as a whole that will facilitate access to particular types of information within the table. For example, it has been convenient to attach a structure keyed on the ID codes for each assignment. This data identifies the locations of each assignment in the table and acts as an inverted file for the data types involved. It permits manipulating the table's contents by specifying an assignment's ID.

The device of the scroll table has proven to be a very effective way of storing the data needed by a scheduler, organizing it in a way that matches its use. We have illustrated this through showing its use in providing convenient overviews of the expected usage of the resource type. Clearly, this is an important feature in making the data available to the manager in a useful format. The device is also useful for a number of other reasons.

The structure of a scroll table fits its role in the system's internal operations. A scheduler's primary function is to respond to requests for the assignment of its resources. These requests typically specify the desired start and end time, and do not name a particular resource. The scheduler must first determine which of its resources are available. It then either makes a selection itself, or refers the decision to the manager, depending on the authority given to it. Once the selection has been made, the assignment can be entered into its data and the assignment returned to the requesting source. The key operation is the determination of the available resources. The format of the scroll table is convenient since it leads to a rapid search algorithm. The column that includes the requested start time can be scanned to determine possible candidates.

For each of the candidates, the required section of the row can then be scanned. The result is a rapid determination of which, if any, resources match the request.

If no resource is available for the requested period, the scheduler may be required to inform the requestor of the nearest assignment available. Again, the scroll table's form is convenient for executing the required search.

The form of a scroll table is convenient for maintaining the consistency of its data. The maintenance of consistency is likely to require that, when certain changes are entered, certain other changes must also occur. These other changes are likely to be in the same row as the original change, but over some following period of time. The structure of the scroll table facilitates locating the cells in which these changes must be made.

The use of a scroll table to hold the data of a scheduler has proven very useful.

B. Specification of the Model

The second major aspect of the schedulers is the way they encode and use the resource models, which contain the knowledge that controls their operations.

The dominant requirement is that the resource model be encoded as a distinct structure that can be modified easily by the manager. This is necessary so that the system can be adapted to adapted to changing requirements and conditions. It is unlikely that any system can be designed, ab initio, in a way that continues to respond to managerial needs. The ability to modify the system as needs change, or to meet exceptional situations, is aided greatly by encoding the model as a distinct structure.

Also, in entering changes to adapt the system to new requirements, or to meet exceptional circumstances, the user should be able to concentrate on the substance of the changes he desires. As far as possible, he should be able to deal with the knowledge directly, without being involved with the details of how it will be used by the system.

These considerations imply that the functions used by a scheduler should be quite general ones, capable of executing their jobs in accordance with a wide variety of resource models. The resource model should be used to specialize these generalized functions to the particular requirements of the resource type.

A resource model can be viewed as defining an abstract machine, or automaton. The states of this machine include all permissible configurations of the data. It is in this sense that the model defines what is meant by self-consistency. The model, then, specifies what state-transitions are permitted. When new data creates an inconsistency, the model then specifies, directly or indirectly, what other changes must be made to re-establish consistency. For this purpose, the model identifies and labels the categories of information that must be handled, and names the functions to be called under specified circumstances. These components of a resource model are discussed shortly.

As an example, Table 4 shows an abbreviated version of the resource model for the pilot scheduler of ACS.1. It includes four entry types, MISSION, MISSION-REST, SICK and TRAINING. MISSION denotes when a pilot is assigned to a mission. MISSION-REST identifies a rest period following assignment. SICK describes a pilot on sick leave and unavailable for assignment. TRAINING covers class-room study and exemplifies activities with lower priority than an assignment, but which can be scheduled around assignments. The full resource model must account for a number of other situations that can being on leave or attached or detached from the squadron. The components shown have been selected to illustrate important variations in the applicable constraints.

The data structures under each of the names of the entry types is called the "model" for that entry type. The term "resource model" is reserved for the entire collection of models that apply to each of the entry types recognized for the resources of the type handled by the scheduler.

Table 4
PART OF THE RESOURCE MODEL FOR PILOTS

```
MODELS:
  MISSION:
      STATE-NAME:  ASG.
      LEVEL:  5.
      ID-LABEL:  MISSION-ID.
      CLASS-LABEL:  FPC.
      POST-ENTRY:  MISSION-REST.
      E-LIST:
          TYPE:  1.
          WATCH-DEMON:
              WATCH.1:
                  CALL-FUNCTION:  DIALOG.1.
  MISSION-REST:
      STATE-NAME:  ASG.RET.
      LEVEL:  3.
      DURATION:  360.
      ID-LABEL:  MISSION-ID.
      E-LIST:
          TYPE:  2.
          WATCH-DEMON:
              WATCH.2:  NIL.
          SET-DEMON:
              WATCH.2:  NIL.
  SICK:
      STATE-NAME:  SICK.
      LEVEL:  8.
      CLASS-LABEL:  TYPE.
      E-LIST:
          TYPE:  1.
  TRAINING:
      STATE-NAME:  TRAIN.
      LEVEL:  3.
      ID-LABEL:  TRAIN-ID.
      CLASS-LABEL:  TOPIC.
      E-LIST:
          TYPE:  3.
          WATCH-DEMON:
              WATCH.3:  NIL.
          SET-DEMON:
              WATCH.3:  NIL.
```

The printout of Table 4 reformats the actual data structure for better readability. The actual structure is a list of lists of lists, etcetera. That is, the values of MODELS is a list of lists, one for each entry type. For example, the value of MISSION is a list, most of whose elements are dotted pairs indicating property-value pairs. Within MISSION, the value of STATE-NAME is ASG, for "assigned." Each entry type also has a property named E-LIST (for "entry list"), whose value is a list that gives technical information about the entry type. In particular, it identifies the type number and the function names to be used as demons or called by demons, as described later.

Each entry type has a specified value of the property STATE-NAME which identifies the entry type in the table. It is this value that is printed out in Tables 1 and 2, although the format of the print function permits specifying other properties. The table itself defines the default state, which, in this case, is AVAIL (for "available") that is entered when no other entry has been made.

Each entry type has a specified value of the property LEVEL. This encodes the constraint that specifies when data can be superceded by a later entry. The rule generally used is that a new entry can not displace existing data unless its level is higher. However, there are conditions discussed later, when entry should be permitted if the levels are equal. The functions used for entering data permit the setting of a flag that will permit equality.

The use of a numeric level to specify the entrance constraint is somewhat limiting. Should it prove necessary to use a more complex rule, there are other ways to encode them. For example, the set of states that a given entry type can displace could be listed directly. However, the specification of a level for each entry type, with 0 for the default condition, has proven sufficient so far.

Some entry types, such as MISSION, specify a value of ID-LABEL. This value is used as the property name under which an identifying code can be included in the data. For example, if a mission is identified as number M1, the data for it will include the dotted pair (MISSION-ID . M1). The value of CLASS-LABEL is used similarly. In the entry type MISSION, this value is FPC, for "flight purpose code." One could use this label not only for entering the applicable code, but also for other information, even a textual description of the flight or its circumstances.

When present, the value of POST-ENTRY specifes another entry type to be used after the completion of the given entry. For example, in MISSION, it expresses the policy that, following assignment to a mission, a pilot should be given a specifed rest period if possible. The duration is specified as the value of DURATION in the model for MISSION-REST. This constraint is a weak one, it can be violated if

circumstances require. This is reflected in the low value of its level and in the fact that its TYPE is #2, the significance of which is discussed next.

The value of E-LIST, for entry list, is itself a list which identifies some of the technical aspects of the entry type. The value of TYPE in the E-list designates the basic characteristics of the entry type. Type 1, as in MISSION, requires that the entry be made in its entirety over its specified interval, or not at all. It makes no sense, for example, to assign a pilot for only part of a mission.

Other types also exist. For example, MISSION-REST is type 2, indicating that it is to be made continuously from the start and as far as possible, considering the existing data and its level, up to the specified duration. Type 3 permits a discontinuous entry until the required duration has been obtained. For example, it is used for classroom training. Other types could be defined as required to describe other needs.

The value of WATCH-DEMON in the E-list names a function to be used in a demon*, as described in the next subsection. The value of CALL-FUNCTION is also the name of a function, and is used by the function named under WATCH-DEMON, as is also described later. It is sufficient to say, here, that it determines what the scheduler does if the later entry of data creates a conflict with the assignment of a given pilot to a given mission. The function named in Table 3 creates a dialog with the commander so that he can determine how the conflict is to be resolved. By changing what function is named, the the scheduler can be given the authority to reschedule the mission to a different pilot, and told how to select the substitute.

In some cases, the E-list also includes a property, SET-DEMON, which names a function to be used in a demon. This entry may also specify a value of CALL-FUNCTION to be used by the set demon. The differences between watch and set demons, and the complementary roles

---

* A demon is a structure that is attached to one or more data elements. It contains a precondition and a function together with means of setting the arguments of the function when it is called. If the data is changed in a way that satisfies the precondition, the function is called with the required arguments and executes what it has been programmed to do. The demon is then said to have been "fired".

To be precise, this describes "write demons." There may also be "read demons" that may be fired on reading the data items. We have not used read demons, and will use the term "demon" to refer only to write demons.

they play in certain situations, are described later. Briefly, a watch demon is attached to data entries that have been made successfully, while set demons are attached to data that has blocked a desired entry. The same function may be used in both demons, as is illustrated in Table 4 for MISSION-REST and TRAINING. However, the demons are different, both because they are attached to different data elements, and because they use different preconditions. They are specified specified to permit using different functions, should this be needed.

The resource model is encoded, then, into a data structure as described. This structure is attached to the scroll table as a whole, and so is identified closely with the scroll table of the scheduler. This format is convenient and has the flexibility required to handle a wide variety of specifications.

The specification of the resource model in the format described not only meets the requirement that it shall be explicit and accessible. It also facilitates modifying the resource model to meet changing conditions, policies, or needs. The key elements of the resource model are named explicitly, so that their values can be changed easily. Changing them will switch the behavior of the functions that use the resource model, or will change the parameters they use. The resource model appears to be a flexible, powerful, and convenient way to encode the knowledge used by the schedulers.

C. Self-Consistency in a Scheduler

In the previous section, we mentioned demons as the means used to enforce the continuing self-consistency of the data in a scroll table. The way they do this is described in detail later. Here, we are concerned with determining what is the self-consistency that is required. There is need for a precise definition of the term.

The problem of consistency arises because new data may be entered at any time that will invalidate previous plans. For example, if a pilot becomes sick, or an aircraft is found to need maintenance, this data will force reconsideration of plans to use that pilot or aircraft. The plans must either be cancelled or revised to make them consistent with the new situation.

The consistency that is required we call "retroactive consistency." Its abstract definition is as follows:

Let P1 and P2 be procedures that seek to enter data into a scheduler. Whether or not they succeed depends on the rules used by the scheduler and the data that is already present in it. Let P1 and P2 be executed in either order. Then let any data remaining from P1 be deleted from the table. The table should then be the same as it would have been, had one of the following three sequences occurred:

* P1 is executed and then its data deleted, without execution of P2,

* First P1 is executed, then its data is deleted, then P2 is executed, and finally its data is deleted,

* P1 is executed, then its data is deleted, then P2 is executed.

In all cases, the operations with P1 are completed first. Then nothing is done, or P2 is first executed and then deleted, or P2 is excuted.

Note that the deletion of data entered by a process, P, is not a true inverse. Its entry may have forced cancellation of some previous entry which is not recovered when the data of P is deleted. Hence, the table may not be returned to the same state as it was in prior to its execution. It is for this reason that, in all cases, P1 is first executed and then its data is deleted and, in case b, the same thing is then done with P2.

In effect, the deletion of the P1 data is required to have the same effect as if either P2 were never executed, or P2 were executed afterwards, and its data then either deleted or not. It is for this reason that we call it "retroactive consistency."

In the case where P2 is a type 1 entry, where the entry must be made in its entirety or not at all, any of the three cases can apply. If P1 has precedence, and if it interferes with P2, involving some of the same locations in the scroll table, the prior entrance of P1 will block P2, so that case a applies. If P2 is entered first, the subsequent entry of P1 will cause the cancellation of the P2 data, leading to case b. (It is not case a since the entry and deletion of the P2 data may cause changes elsewhere in the table.) Finally, if P2 has precedence, then case c will apply, whether there is interference or not.

For types 2 and 3, case c applies. Type 2, it will be recalled, makes the entry as far as possible from the specified start time until the desired duration has been obtained, or until the entry is blocked. Type 3 permits a discontinuous entry starting from the specified start time until the desired duration has been accumulated. In both cases, the deletion of the P1 data may open the possibility of a more complete P2 entry. Retroactive consistency requires that full advantage be taken of this possibility, when it occurs.

For example, suppose MISSION-REST specifies a duration of six hours. Suppose, however, that a pilot is scheduled for sick leave starting four hours after the end of a mission. The state, ASG.RET, can ae entered only for the four hours. If, later, the sick leave is cancelled, it is required that the entry of ASG.RET be extended to the full six hours set by policy. The effect is as if the entry of MISSION-REST were made after, rather than before, the cancellation of the sick leave.

Suppose, again, the MISSION-REST entry were executed first for the full six hours. Suppose the sick leave is entered later, reducing the extent of ASG.RET to four hours. If the sick leave is then cancelled, the extent of ASG.RET must then be reinstated for the full six hours.

There are two ways a later data entry can affect a prior entry. It can override data that is present, or it make cells availble that were not available before. The former case is handled by watch demons, which have the responsibility of recognizing when a later entry creates a conflict. The latter case is handled by set demons, which have the responsibility of recognizing when a later change creates an opportunity to improve an earlier entry. Watch demons, when set by a process P, are attached to the cells in which P makes an entry. Set demons, when created by P, are attached to a cell or cells in which P attempts to make an entry, but is blocked by data already present.

The abstract description definition of retroactive consistency suggests what the principle action of a watch or set demon should be. Put somewhat over-simply, it should either cancel the original entry operation or re-execute it. In the latter case, it needs to be able to overwrite data that may remain from the original entry, since some of the parameters of this data may be changed. (For example, if the data includes the actual range of the entry, this will now be different.) In the latter case also, it may need to determine where data from the original entry may be that is not overwritten, and delete this data. The primary action is still either cancellation or re-execution.

An important principle in the design and use of watch and set demons, and to complementary pairs of them, appears to be what might be called the "self-destruct capability". This states that, when a demon is fired, it should have all information necessary for its removal available to it, and for the removal of its complement if any. The importance of this principle is that, when a watch or set demon is fired, it may modify data to which it is attached. To avoid refiring the demon, the first action of the demon when fired must be the elimination of all its occurences, and of all occurences of its complement. The self-destruct property allows this to be done.

Techniques for constructing and using demons, and for the convenient implementation of the self-destruct property, are discussed later.

D. Alert and System-Initiated Actions

The final aspect is the way system-initiated actions can be programmed, including the issuance of alert messages.

The requirement can be stated somewhat abstractly by stating that, when some certain precondition is met, certain actions should be taken. This statement is a description of a demon, and demons are the device used for obtaining system-initiated actions and messages. However, the demons that may be used here may be rather different from the watch and set demons used for maintaining retroactive consistency in the table.

The first difference is that preconditions used for the demon may be quite complex. The precondition for a watch demon is often empty, the demon is to be fired whenever data in the cell being watched is changed. The one on a set demon is likely to be non-empty but still simple, the demon is to be fired if the value of level in the data in a cell is made less than some value. By contrast, an alert demon may be required to initiate an alert message only if some condition is reached that requires some fairly extensive computations.

For example, the commander of an air squadron may want to be alerted if the number of aircraft available for flying a particular type of mission should fall below what he defines as an adequate reserve. The duration of the mission type may be specified. The precondition on the alert message requires evaluating not only how many aircraft are available for a mission at any given time, but whether they are available for the required period of time, starting at any given time.

One consequence of the possible complexity of the precondition for the specified action is that it may not be used as the precondition of the demon itself. It may be more convenient to let the demon be fired on any change of the data that might interfere with flying a mission (i.e., that raised the level of the data above that of assignment to a mission) and to incorporate the rest of the evaluation process into the function called by the demon. The precondition of the demon need not be that of the system-initiated action being implemented by the demon.

A second difference from watch and set demons is that the data involved is not confined necessarily to a single cell. In ACS.1, we have made provision for four types of demons, depending on the data structures they are attached to. Watch and set demons are examples of element demons, attached to particular cells in the table. We have also made provision for column, row and table demons. Column demons are attached to columns and are checked whenever an entry is made in that column. Row demons are attached to rows, and checked whenever an entry is made to the given row. Table demons are attached to the table as a whole, and checked on any entry to the table.

The possibility of demons whose scope is defined other than a cell, column, row or the entire table could be considered. We could also consider demons on data other than what is in the table, such as cumulative data attached to a row or the table as a whole. We have not found the need for these other possibilities, however. It appears that all system-initiated actions, which depend on the evaluation of information held totally within a given scheduler, can be handled with the varieties of demons that have been defined.

The third difference from watch and set demons is in the variety of responses that may be required. A watch or set demon, as discussed, must have the effect of reexecuting the process that caused their creation. Hence the act of creation defines their operation, at least in the abstract sense. By contrast, a demon that drives a system-initated process may be required to do whatever the user specifies. In consequence, the function used by the demon cannot be standardized. The structure of the demon itself, and the way in which its preconditions are tested and the demon fired, can be standardized, but not the function or its behavior. However, the problem of creating the desired demon is reduced to defining the function that will execute the specified action, given the environment that is specified by the conditions that will cause the demon to be fired.

While the specific demons used for alert and system-initiative purposes cannot be specified in any general way, ACS.1 does provide a flexible and convenient environment in which they can be defined as the need arises.

V FUNCTION ORGANIZATION

The specific functions defined for the schedulers of ACS.1 are given and described in the next seven sections. Each section is concerned with one group of functions, although these may be divided into subgroups for clarity. The groups have a generally hierarchical organization according to the levels named in Table 5. Each group of functions may use functions in the groups above it in Table 5. The numbers before the titles in that table are the corresponding section and subsection numbers; so, that table 5 serves as an index to the following material.

Table 5
OGANIZATION OF THE SCHEDULER FUNCTIONS

VI MISCELLANEOUS CONVENIENCE FUNCTIONS

VII TABLE FUNCTIONS

- A. Table Creation
- B. Row Naming
- C. Scrolling
- D. Column Creation
- E. Row Modification and Retrieval

VIII PRIMARY DATA FUNCTIONS

- A. Data Entry
- B. Data Retrieval and Display

IX MODEL FUNCTIONS

- A. Creation and Display of a Model
- B. Modification of a Model

X DEMON HANDLING

- A. Attachment of Demons
- B. Removal of Demons
- C. Self-Destruct Property

XI TOP LEVEL FUNCTIONS

- A. Top Level Data Entry
- B. Data Cancellation

XII DEMON AND DIALOG FUNCTIONS (WATCH AND SET DEMONS)

Note that all of the functions are defined in INTERLISP, so that the usual INTERLISP functions represent a bottom level not shown in Table 5.

In the following sections, the actual functions in each group and subgroup are given at the end of the corresponding section or subsection. The body of the section is textual material that describes the functions and discusses some of the reasons for the choices that have been made.

VI MISCELLANEOUS CONVENIENCE FUNCTIONS

This and the following sections describe the functions used by ACS.1 for creating, manipulating and using scroll tables. ACS.1 is written in INTERLISP and operates under TOPS.10.

There are a number of functions of general utility in ACS.1. One set are those used for manipulating A-lists, adding, deleting or changing property-value pairs, or for retrieving by property name information stored on an A-list. The names of these functions are preceded by "A." as in A.GETP. The type of operation is indicated by the core of the name, PUT, GETP, ADDPROP, REMPROP or REMVAL. The PUT functions put the specified value under the property name, overwriting any previous value, if any. The GETP functions recover the value of the named property, executing an associative retrieval (hence the name, A-list, for associative list.) The ADDPROP functions are similar to the PUT functions, except that the value of the property is expected to be a list of values, so that overwriting is not required. If the property is being added, the value is entered as a list of one term. If the property already exists, the new value is added to the existing list of values. The REMPROP functions remove the named property, and its value, from the A-list, except that it cannot anihilate the A-list. The REMVAL functions remove the indicated value from a list of values of the property.

The functions that have ".S" on the end, such as A.GETP.S, use SASSOC instead of ASSOC. In locating the property, this forces the use of EQUAL instead of EQ. Where the property name is a number, as may be the case when values are stored under an ID number, EQ may fail when we would want it to succeed. A.GETP.S will succeed when A.GETP might not.

Finally, the functions that have "#" on the end, such as A.GETP# or A.GETP.S#, take a list of property names. The a-list is searched recursively, using successive elements of the property list, until the property list is reduced to a single element. The function then is applied using the remaining element as the property. The function fails if any property on the list is not in the value of the preceding property. If it fails, it does nothing and returns NIL.

The function, CONVERT.TIME, is used to reformat time, given as an integer which is the number of minutes from some starting time. It expresses it as hours:minutes. The second part is forced to have two digits by adding a zero to the front if necessary. The first part similarly is forced to have at least two digits.

DUMMY is a trivial function that always returns T. It is used in the precondition of demons where it is desired that the demon shall always be fired whenever the data elements to which it is attached are changed.

The function, IREMAINDER1, is used in manipulating the index array, described later, of a scroll table. It returns X modulo Y as a number in the range 1 through Y, rather than the more usual residue from 0 through (Y - 1).

The function, MAPPRINQ, is useful for fairly complicated printouts. LST is a list handled in order. If an element on the list is TERPRI, (TAB <n>), or (RPTQ <n> <function>) it is executed. If it is a string in quotation marks, it is printed. If it is an atom, its value is printed. Otherwise, it is evaluated and its value printed.

The function, PRINT.LIST, is used to print lists in a convenient format. For example, Table 4 could be printed using this function. In general, it uses indentation to indicate depth within the list, adding a colon after the first element of each list. It assumes, in other words, that the list is an A-list, so that the first element of any list or sublist is a property name.

The function, SUMMARY, is useful for printing complicated and lengthy lists where only a general overview is desired. It sets the print level to that indicated, or to 3 if no value given, prints the list, and then returns the print level to its usual value of 1000. Parts of the list that are at a level greater than the print level are then indicated by "&" without elaboration.

The functions, NEXTLOWER and NEXTHIGHER, convert an arbitrary time to a value that is consistent with the quantized time used in a scroll table. The table specifies a start time and an interval. The start time of any column of the table is required to be the sum of the start time and an arbitrary integral non-negative number of intervals. NEXTLOWER returns the start time of the column, if it exists, that contains the specified time. If the time is before the start time, it returns the start time. NEXTHIGHER returns the end time of the column, if one exists, that contains the time. If the time is the start time of a possible column, the time itself is returned. If the time is NIL or not a number, it returns "INDEF."

The function, SIM.CLOCK, updates the global variable S.CLOCK which is the simulated time used by the system.

The function, LAST.ENTRY, takes a list of dotted pairs whose elements are numbers, and returns the largest of the second members of the pairs. If the second member of any pair is nonnumeric, it returns T. It is used on a list of intervals to determine the end

of the latest interval. Where the list of intervals locates a discontinuous entry, it provides a bound on the entry.

Three other functions are used to handle time intervals. CHECK.TIME takes an interval and compares it with the system variable, S.CLOCK, the simulated time of the system. If S.CLOCK is in the interval, it is returned. If the interval is after S.CLOCK, its start time is returned. If it is before S.CLOCK, NIL is returned. If the end of the interval is not specified, or not a number, the interval is assumed to be open-ended, and either S.CLOCK or the start of the interval is returned. It is used to find a location in an entry which is currently in the scroll table, where scrolling may have occured since the entry was made.

CHECK.TIME.LST takes a list of intervals expressed as dotted pairs. It returns CHECK.TIME on the first interval, if any, for which its value is non-NIL. It is used to find a time within the current scope of the scroll table in the list of intervals. If an entry has been made in a possibly discontinuous way, and the table may have been scrolled since the entry, it is used to locate one occurence of the entry.

Finally, CHECK.LST takes a list of dotted pairs describing intervals of time and compares each to S.CLOCK. If the start of an interval is at least as large as S.CLOCK, it is retained. If the end is before S.CLOCK, the interval is discarded. Otherwise, the interval is changed to (S.CLOCK . <end>), where the end may be "INDEF." The purpose is to make the list of intervals consistent with S.CLOCK. It is used on entries that may be discontinuous, when the scroll table may have been scrolled since the entry was made.

These functions are defined as follows:

```
(A.ADDPROP
  [LAMBDA (A.LIST PROP VAL FLG)
    (PROG (T0 T1)
          (SETQ T0 (ASSOC PROP A.LIST))
          (SETQ T1 (CDR T0))
          (COND
            ((NULL T1)
              (A.PUT A.LIST PROP (LIST VAL))
              VAL)
            [(LISTP T1)
              (COND
                (FLG (RPLACD T0 (CONS VAL T1))
                     (CDR T0))
                ((RPLACD T0 (NCONC1 T1 VAL))
                  (CDR T0]
            ((NULL FLG)
              (RPLACD T0 (NCONC (LIST T1)
                                (LIST VAL)))
              (CDR T0))
            ((RPLACD T0 (CONS VAL (LIST T1)))
              (CDR T0])

(A.ADDPROP#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.ADDPROP A.LIST (CAR PROP.LIST)
                   VAL))
      (T (A.ADDPROP# (A.GETP A.LIST (CAR PROP.LIST))
                     (CDR PROP.LIST)
                     VAL])

(A.ADDPROP.S
  [LAMBDA (A.LIST PROP VAL FLG)
    (PROG (T0 T1)
          (SETQ T0 (SASSOC PROP A.LIST))
          (SETQ T1 (CDR T0))
          (COND
            ((NULL T1)
              (A.PUT.S A.LIST PROP (LIST VAL))
              VAL)
            [(LISTP T1)
              (COND
                (FLG (RPLACD T0 (CONS VAL T1))
                     (CDR T0))
                ((RPLACD T0 (NCONC1 T1 VAL))
                  (CDR T0]
            ((NULL FLG)
              (RPLACD T0 (NCONC (LIST T1)
                                (LIST VAL)))
              (CDR T0))
            ((RPLACD T0 (CONS VAL (LIST T1)))
              (CDR T0])
```

```
(A.ADDPROP.S#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.ADDPROP.S A.LIST (CAR PROP.LIST)
                     VAL))
      (T (A.ADDPROP.S# (A.GETP.S A.LIST (CAR PROP.LIST))
                       (CDR PROP.LIST)
                       VAL])

(A.GETP
  [LAMBDA (A.LIST PROP)
    (CDR (ASSOC PROP A.LIST])

(A.GETP#
  [LAMBDA (A.LIST PROP.LIST)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.GETP A.LIST (CAR PROP.LIST)))
      (T (A.GETP# (A.GETP A.LIST (CAR PROP.LIST))
                  (CDR PROP.LIST])

(A.GETP.S
  [LAMBDA (A.LIST PROP)
    (CDR (SASSOC PROP A.LIST])

(A.GETP.S#
  [LAMBDA (A.LIST PROP.LIST)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.GETP.S A.LIST (CAR PROP.LIST)))
      (T (A.GETP.S# (A.GETP.S A.LIST (CAR PROP.LIST))
                    (CDR PROP.LIST])

(A.PUT
  [LAMBDA (A.LIST PROP VAL)
    (PROG (TO)
          [COND
            ((SETQ TO (ASSOC PROP A.LIST))
              (RPLACD TO VAL))
            ((NCONC1 A.LIST (CONS PROP VAL]
          (RETURN VAL])

(A.PUT#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.PUT A.LIST (CAR PROP.LIST)
               VAL))
      (T (A.PUT# (A.GETP A.LIST (CAR PROP.LIST))
                 (CDR PROP.LIST)
                 VAL])
```

```
(A.PUT.S
  [LAMBDA (A.LIST PROP VAL)
    (PROG (X)
          [COND
            ((SETQ X (SASSOC PROP A.LIST))
              (RPLACD X VAL))
            (T (NCONC1 A.LIST (CONS PROP VAL]
          (RETURN VAL])

(A.PUT.S#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.PUT.S A.LIST (CAR PROP.LIST)
                 VAL))
      (T (A.PUT.S# (A.GETP.S A.LIST (CAR PROP.LIST))
                   (CDR PROP.LIST)
                   VAL])

(A.REMPROP
  [LAMBDA (A.LIST PROP)
    (PROG (SUB.LIST)
          (COND
            ((SETQ SUB.LIST (ASSOC PROP A.LIST))
              (COND
                ((EQUAL A.LIST (LIST SUB.LIST))
                  (RPLACD SUB.LIST NIL))
                (T (DREMOVE SUB.LIST A.LIST)))
              (RETURN PROP])

(A.REMPROP#
  [LAMBDA (A.LIST PROP.LIST)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.REMPROP A.LIST (CAR PROP.LIST)))
      (T (A.REMPROP# (A.GETP A.LIST (CAR PROP.LIST))
                     (CDR PROP.LIST])

(A.REMPROP.S
  [LAMBDA (A.LIST PROP)
    (PROG (SUB.LIST)
          (COND
            ((SETQ SUB.LIST (SASSOC PROP A.LIST))
              (COND
                ((EQUAL SUB.LIST (LIST SUB.LIST))
                  (RPLACD SUB.LIST NIL))
                (T (DREMOVE SUB.LIST A.LIST)))
              (RETURN PROP])
```

```
(A.REMPROP.S#
  [LAMBDA (A.LIST PROP.LIST)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.REMPROP.S A.LIST (CAR PROP.LIST)))
      (T (A.REMPROP.S# (A.GETP.S A.LIST (CAR PROP.LIST))
                       (CDR PROP.LIST])

(A.REMVAL
  [LAMBDA (A.LIST PROP VAL)
    (PROG (X)
          (SETQ X (ASSOC PROP A.LIST))
          (COND
            [(EQUAL X (LIST PROP VAL))
              (COND
                ((CDR A.LIST)
                  (A.REMPROP A.LIST PROP))
                (T (RPLACA A.LIST (LIST PROP]
            (T (DREMOVE VAL X)))
          (RETURN VAL])

(A.REMVAL#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.REMVAL A.LIST (CAR PROP.LIST)
                  VAL))
      (T (A.REMVAL# (A.GETP A.LIST (CAR PROP.LIST))
                    (CDR PROP.LIST)
                    VAL])

(A.REMVAL.S
  [LAMBDA (A.LIST PROP VAL)
    (PROG (X)
          (SETQ X (SASSOC PROP A.LIST))
          (COND
            [(EQUAL X (LIST PROP VAL))
              (COND
                ((CDR A.LIST)
                  (A.REMPROP.S A.LIST PROP))
                (T (RPLACA A.LIST (LIST PROP]
            (T (DREMOVE VAL X)))
          (RETURN VAL])

(A.REMVAL.S#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((NULL (CDR PROP.LIST))
        (A.REMVAL.S A.LIST (CAR PROP.LIST)
                    VAL))
      (T (A.REMVAL.S# (A.GETP.S A.LIST (CAR PROP.LIST))
                      (CDR PROP.LIST)
                      VAL])
```

```
(CONVERT.TIME
  [LAMBDA (TIME)
    (PROG (X Y STR1 STR2)
          (SETQ X (FIX (IQUOTIENT TIME 60)))
          (SETQ Y (IREMAINDER TIME 60))
          (COND
            ((LESSP X 10)
              (SETQ STR1 (CONCAT 0 X)))
            (T (SETQ STR1 X)))
          (COND
            ((LESSP Y 10)
              (SETQ STR2 (CONCAT 0 Y)))
            (T (SETQ STR2 Y)))
          (RETURN (CONCAT STR1 ":" STR2])

(DUMMY
  [LAMBDA (NIL) T])

(IREMAINDER1
  [LAMBDA (X Y)
    (COND
      ((EQ X 0)
        Y)
      (T (ADD1 (IREMAINDER (SUB1 X)
                           Y])

(MAPPRINQ
  [NLAMBDA (LST)
    (MAPC LST (FUNCTION (LAMBDA (#X)
              (COND
                ((EQUAL #X (QUOTE TERPRI))
                  (TERPRI))
                ((STRINGP #X)
                  (PRIN1 #X))
                ((ATOM #X)
                  (PRIN1 (EVAL #X)))
                ((EQUAL (CAR #X)
                        (QUOTE TAB))
                  (TAB (CADR #X)))
                [(EQUAL (CAR #X)
                        (QUOTE RPTQ))
                  (RPTQ (CADR #X)
                        (EVAL (CADDR #X]
                (T (PRIN1 (EVAL #X])
```

```
(PRINT.LIST
  [LAMBDA (LST N)
    (COND
      ((NULL N)
        (PRINT.LIST LST 0))
      ((NULL LST)
        NIL)
      [(ATOM (CAR LST))
        (COND
          ((ATOM (CDR LST))
            (TAB N)
            (PRIN1 (CAR LST))
            (PRIN1 ":  ")
            (PRIN1 (CDR LST))
            (PRIN1 ".")
            (TERPRI))
          ([AND (NULL (CDDR LST))
                (NOT (LISTP (CDR LST]
            (TAB N)
            (PRIN1 (CAR LST))
            (PRIN1 ":  ")
            (PRIN1 (CADR LST))
            (PRIN1 ".")
            (TERPRI))
          (T (TAB N)
             (PRIN1 (CAR LST))
             (PRIN1 ":")
             (TERPRI)
             (PRINT.LIST (CDR LST)
                         (IPLUS N 4]
       (T (PRINT.LIST (CAR LST)
                      N)
          (PRINT.LIST (CDR LST)
                      N])

(SUMMARY
  [LAMBDA (LST PRINT.LEVEL)
    (PROG (NIL)
          (COND
            ((NULL PRINT.LEVEL)
              (SETQ PRINT.LEVEL 3)))
          (PRINTLEVEL PRINT.LEVEL)
          (PRINT LST)
          (PRINTLEVEL 1000]))
```

```
(NEXTLOWER
  [LAMBDA (TABLE.NAME TIME)
    (PROG (S.TIME)
          (SETQ S.TIME (GETP TABLE.NAME (QUOTE START)))
          (COND
            ((OR (NULL TIME)
                 (LESSP TIME S.TIME))
              (RETURN S.TIME))
            (T (RETURN (IPLUS TIME (IMINUS (IREMAINDER
                                             TIME
                                             (GETP TABLE.NAME
                                                   (QUOTE INTERVAL])

(NEXTHIGHER
  [LAMBDA (TABLE.NAME TIME)
    (PROG (INTERVAL)
          (SETQ INTERVAL (GETP TABLE.NAME (QUOTE INTERVAL)))
          (COND
            ((NOT (NUMBERP TIME))
              (RETURN (QUOTE INDEF)))
            [(LESSP TIME (GETP TABLE (QUOTE START)))
              (RETURN (NEXTHIGHER TABLE (GETP TABLE (QUOTE START]
            ((ZEROP (IREMAINDER TIME INTERVAL))
              (RETURN TIME))
            (T (RETURN (IPLUS TIME INTERVAL
                              (IMINUS (IREMAINDER TIME INTERVAL])

(SIM.CLOCK
  [LAMBDA (CLOCK.TIME)
    (PROG (X)
          (COND
            ((NULL CLOCK.TIME)
              (RETURN S.CLOCK))
            (T (SETQ X S.CLOCK)
               (SETQ S.CLOCK CLOCK.TIME)
               (RETURN X])

(LAST.ENTRY
  [LAMBDA (LST)
    (PROG (TRIAL)
          [MAPC LST (FUNCTION (LAMBDA (X)
                    (COND
                      ((NOT (NUMBERP (CDR X)))
                        (RETURN T))
                      ((OR (NULL TRIAL)
                           (GREATERP (CDR X)
                                     TRIAL))
                        (SETQ TRIAL (CDR X]
          (RETURN TRIAL])
```

```
(CHECK.TIME
  [LAMBDA (TABLE TIME END)
    (COND
      ((GREATERP TIME S.CLOCK)
        TIME)
      ((NOT (NUMBERP END))
        S.CLOCK)
      ((GREATERP S.CLOCK END)
        NIL)
      (T S.CLOCK])

(CHECK.TIME.LST
  [LAMBDA (TABLE TIME.LST)
    (PROG (NIL)
          (MAPC TIME.LST (FUNCTION (LAMBDA (X)
                    (COND
                      ((GREATERP (CAR X)
                                 S.CLOCK)
                        (RETURN X))
                      [(NOT (NUMBERP (CDR X)))
                        (RETURN (CAR S.CLOCK (QUOTE INDEF]
                      ((GREATERP S.CLOCK (CDR X)))
                      (T (RETURN (CAR S.CLOCK (CDR X])

(CHECK.LST
  [LAMBDA (LST)
    (PROG (NEW.LST)
          [MAPC LST (FUNCTION (LAMBDA (X)
                    (COND
                      ((GREATERP (CAR X)
                                 S.CLOCK)
                        (SETQ NEW.LST (NCONC1 NEW.LST X)))
                      [(NOT (NUMBERP (CDR X)))
                        (SETQ NEW.LST (NCONC1 NEW.LST
                                              (CONS S.CLOCK
                                                    (QUOTE INDEF]
                      ((GREATERP S.CLOCK (CDR X)))
                      (T (SETQ NEW.LST (NCONC1 NEW.LST (CONS S.CLOCK
                                                             (CDR X]

          (RETURN NEW.LST])
```

VII TABLE FUNCTIONS

The functions described in this section are those used for the direct manipulation of the scroll table. That is, they are the functions that set up the scroll table and that do the operations on it that are independent of the resource model. Therefore, they are not generally top level functions, but are those used by the top level ones.

A. Table Creation

A scroll table is created by the function MAKE.TABLE. The parameters given it are the name of the scroll table or the scheduler (we do not distinguish between them), the number of rows that are to be made available, the start time of the table, the number of intervals that are to included in the index array as discussed shortly, the size of the interval in minutes, and the default state name. The interval is the length of time covered by each column in the virtual, or top level, view of the scroll table.

The index array, which has not been discussed previously, is a circular list that is used to provide quick access to the early part of the table, through the first n columns of the virtual table, where n is the number of intervals specified as N.INT. Some cell in this array is designated as the starting one. Each cell around the circular list refers to the following interval of time. Each cell contains a pointer to the actual column containings the given time. For example, if the interval size is 30 minutes and n is 24, the index array covers 12 hours after the start time of the table. If the start index is currently #1 and the start time is zero minutes, #2 refers to times from 30 minutes to 59, #3 from 60 to 89 minutes, and so on. If the first column has a start time of 0 and an end time of, say, 90, then both the #1 and #2 cells will point to it.

As stated, the index array provides a quick access to the part of the table it covers. If access is required for a later time, entry must first be made to the first column beyond those covered by the index array, a pointer to which is maintained by the table. From this column, a pointer to the next column can be retrieved. From that column, a pointer to the next column can be obtained. Access to the required column is obtained by moving "hand over hand" in this way until it is found.

The index array is a circular list to take account of scrolling. When the table is scrolled, the start time of the table is changed, the pointer to the starting cell of the index array is changed to the next cell in the array, the cell that is released is given the pointer to the first column beyond the index array, and the pointer to the first

column beyond the index array is changed if necessary. The function that does all this is ST.SCROLL, given later.

The creation of a table is accomplished by entering the various paramteric values into the property list of the table name. At the same time, the required arrays are created, and the pointers to them also entered into the property list of the table name. The table name is the starting point for all operations on, or using, the table.

Note that the immediate result of calling MAKE.TABLE is an empty table. None of the rows have been named. There is only a single column whose start time is the given starting time and whose end time is "INDEF" indicating that its span continues for an indefinite period. No cells exist because no rows are named. The table exists, but it has no content.

MAKE.TABLE is defined as follows:

```
(MAKE.TABLE
  [LAMBDA (TABLE NROWS START N.INT INTERVAL DEFAULT.STATE
                 PRINT.SUPPRESS.FLG)
    (PROG (LST ARR)
          (PUT TABLE (QUOTE DEFAULT.LIST)
               (LIST (CONS (QUOTE STATE.NAME)
                           DEFAULT.STATE)
                     (CONS (QUOTE LEVEL)
                           0)
                     (CONS (QUOTE DEMONS)
                           NIL)))
          (PUT TABLE (QUOTE NROWS)
               NROWS)
          (PUT TABLE (QUOTE START)
               START)
          (PUT TABLE (QUOTE INTERVAL)
               INTERVAL)
          (PUT TABLE (QUOTE N.INT)
               N.INT)
          [PUT TABLE (QUOTE BEYOND.INDEX)
               (SETQ LST (CONS (ARRAY NROWS)
                               (LIST (CONS (QUOTE FORWARD)
                                           NIL)
                                     (CONS (QUOTE BACKWARD)
                                           NIL)
                                     (CONS (QUOTE START)
                                           START)
                                     (CONS (QUOTE END)
                                           (QUOTE INDEF))
                                     (CONS (QUOTE DEMONS)
                                           NIL]
```

```
        (PUT TABLE (QUOTE INDEX.ARRAY.PTR)
             (ARRAY N.INT NIL LST))
        (PUT TABLE (QUOTE START.INDEX)
             1)
        (PUT TABLE (QUOTE FREE.ROW.INDEX)
             1)
        (PUT TABLE (QUOTE A.LIST)
             (LIST (CONS (QUOTE DEMONS)
                         NIL)))
        (PUT TABLE (QUOTE ROWINDEX.TO.ROWNAME)
             (SETQ ARR (ARRAY NROWS)))
        (PUT TABLE (QUOTE RWONAME.TO.ROWINDEX)
             (CONS (HARRAY NROWS)
                   2))
        (SETA ARR NROWS NIL)
        (SETQ NROWS (SUB1 NROWS))
    L   (COND
          ((EQ NROWS 0)
            [COND
              ((NULL PRINT.SUPPRESS.FLG)
                (MAPPRINQ ("Scroll table named " TABLE " created."
                                                 TERPRI]
            (RETURN (CHARACTER 127)))
          ((SETA ARR NROWS (ADD1 NROWS))
            (SETQ NROWS (SUB1 NROWS))
            (GO L])
```

B. Row Naming

Once the table has been made, the rows need to be named. This is accomplished with the function NAME.ROW. It can be called at any time, not only at the creation of the table. Each of the existing comuns is given a new entry for the new row. The value of this entry is the DEFAULT.LIST of the table that was derived from the default state given to MAKE.TABLE originally. The form of DEFAULT.LIST is:

((STATE.NAME . <default state>)(LEVEL . 0)(DEMONS)).

The value of LEVEL, in any entry to the table, indicates what data can displace the given data. That the level is here set to 0 indicates that any data can displace it. The property DEMONS, with NIL value, is included to provide space for any element demons that may be entered later, as discussed later.

Note that NAME.ROW aborts if all the rows have been named. DELETE.ROW, given later, will remove a named row. EXPAND.TABLE, also given later, has the effect of increasing the number of available rows. Hence, the limit on the number of rows that can be named is not critical.

The function, NAME.ROW, is defined as follows:

```
(NAME.ROW
  [LAMBDA (TABLE ROW.NAME PRINT.SUPPRESS.FLG)
    (PROG (FREE.ROW.INDEX COL.PTR)
          (COND
            [(SETQ FREE.ROW.INDEX (GETP TABLE (QUOTE FREE.ROW.INDEX]
            (T [COND
                 ((NULL PRINT.SUPPRESS.FLG)
                   (MAPPRINQ ("No room.  Number of rows in "
                              TABLE " is " (GETP TABLE (QUOTE NROWS))
                              "." TERPRI]
               (RETURN NIL)))
          (PUT TABLE (QUOTE FREE.ROW.INDEX)
               (ELT (GETP TABLE (QUOTE ROWINDEX.TO.ROWNAME))
                    FREE.ROW.INDEX))
          (PUTHASH ROW.NAME FREE.ROW.INDEX (GETP TABLE (QUOTE
                                                 ROWNAME.TO.ROWINDEX)))
          [SETA (GETP TABLE (QUOTE ROWINDEX.TO.ROWNAME))
                FREE.ROW.INDEX
                (CONS ROW.NAME (LIST (LIST (QUOTE DEMONS]
          [SETQ COL.PTR (ELT (GETP TABLE (QUOTE INDEX.ARRAY.PTR))
                             (GETP TABLE (QUOTE START.INDEX]
      L   (SETA (CAR COL.PTR)
                FREE.ROW.INDEX
                (GETP TABLE (QUOTE DEFAULT.LIST)))
          [COND
            ((NULL (A.GETP COL.PTR (QUOTE FORWARD)))
              [COND
                ((NULL PRINT.SUPPRESS.FLG)
                  (MAPPRINQ (ROW.NAME " entered." TERPRI]
              (RETURN (CHARACTER 127]
          (SETQ COL.PTR (A.GETP COL.PTR (QUOTE FORWARD)))
          (GO L])
```

C. Scrolling

Scrolling is accomplished with the function ST.SCROLL. It advances the pointer to the index array. The pointer stored under BEYOND.INDEX is put in the last cell of the index array, which is the one released by the advance of the index array pointer. The BEYOND.INDEX pointer is changed if appropriate. The start time of the table is changed by one interval. This process is repeated until the end time of the first column that remains is greater than S.CLOCK, or is not a number.

Any columns released by this process are simply abandoned. Since no way of accessing these columns remains, their memory space will be recovered on the next garbage collection.

The function, ST.SCROLL, is defined as follows:

```
(ST.SCROLL
  [LAMBDA (TABLE.NAME)
    (PROG (T.START NEWSTART INTERVAL N.INT DURATION INDEX.ARRAY.PTR
                   START.INDEX BEYOND.INDEX COL.PTR COL.START COL.END)
          (SETQ T.START (GETP TABLE.NAME (QUOTE START)))
          (SETQ NEWSTART (SIM.CLOCK))
          [COND
            ((NOT (IGREATERP NEWSTART T.START))
              (RETURN (CHARACTER 127]
          (SETQ INTERVAL (GETP TABLE.NAME (QUOTE INTERVAL)))
          (SETQ N.INT (GETP TABLE.NAME (QUOTE N.INT)))
          (SETQ DURATION (ITIMES N.INT INTERVAL))
          (SETQ INDEX.ARRAY.PTR (GETP TABLE.NAME (QUOTE INDEX.ARRAY.PTR)
                                      ))
          (SETQ START.INDEX (GETP TABLE.NAME (QUOTE START.INDEX)))
          (SETQ BEYOND.INDEX (GETP TABLE.NAME (QUOTE BEYOND.INDEX)))
      L   (SETQ COL.PTR (ELT INDEX.ARRAY.PTR START.INDEX))
          (SETQ COL.START (A.GETP COL.PTR (QUOTE START)))
          (SETQ COL.END (A.GETP COL.PTR (QUOTE END)))
          [COND
            ((AND (NOT (GREATERP COL.START NEWSTART))
                  (OR (NOT (NUMBERP COL.END))
                      (GREATERP COL.END NEWSTART)))
              (A.PUT COL.PTR (QUOTE BACKWARD)
                     NIL)
              (RETURN (CHARACTER 127]
          (A.PUT (CDR COL.PTR)
                 (QUOTE START)
                 (IPLUS T.START INTERVAL))
          (SETQ T.START (IPLUS T.START INTERVAL))
          (SETA INDEX.ARRAY.PTR START.INDEX BEYOND.INDEX)
          (PUT TABLE.NAME (QUOTE START)
               T.START)
          (COND
            ((EQ (IPLUS T.START DURATION)
                 (A.GETP BEYOND.INDEX (QUOTE END)))
              (SETQ BEYOND.INDEX (A.GETP BEYOND.INDEX (QUOTE FORWARD)))
              (PUT TABLE.NAME (QUOTE BEYOND.INDEX)
                   BEYOND.INDEX)))
          (PUT TABLE.NAME (QUOTE START.INDEX)
               (SETQ START.INDEX (IREMAINDER1 (ADD1 START.INDEX)
                                              N.INT)))
          (GO L])
```

D. Column Creation and Access

The function that creates a new column is ST.COL. It determines whether the desired column is within the range of the index array and calls ST.COL.INNER or ST.COL.OUTER, accordingly.

The basic operation of these functions starts by finding the column that contains the time at which a new column is to start. Call this column C1. Suppose the next column is C2. Initially, the forward pointer of C1 indicates C2, and the backward pointer of C2 indicates C1. A new column, say C3, is created with its forward pointer indicating C2 and its backward pointer indicating C1. The forward pointer of C1 and the backward pointer of C2 are then changed to indicate C3. The columns now are linked properly. The end time of C1 and the start and end times of C3 are entered appropriately. If C3 is within the range of the index array, its entries are changed to match the new column. Finally, the cells in C3, its values in the active rows, are made to duplicate those of C1. The new column is now available for the entry of data as described later.

If a column already exists at the indicated time, nothing is done. Hence the function can be called if the creation of a column may be needed, without any check that it actually is required.

Note that all the data in the new column is an exact copy of the origonal column. This includes the data in the cells for all named rows, including any demons that may have been placed on it. It also includes any data, including demons, that may have been attached to the origonal column as a whole.

Two other functions are concerned directly with the columns. GET.COLS returns a list of the start times of all columns created that cover the interval specified on the call of the function. If the initial time is not given, it is taken as the start time of the table. If the end time is not given, it is taken as indefinite. Hence (GET.COLS <table name>), without specifying either time, returns a list of all columns that currently exist in the table.

The other function is GET.COL.PTR which returns the pointer to the column that covers the given time. A word of warning is in order, however. If this function is called at the top level, the resultant printout is recursive because of the presence of both forward and backward pointers in it. Therefore, precautions must be taken to limit the printout.

The functions are defined as follows:

```
(ST.COL
  [LAMBDA (TABLE.NAME TIME)
    (PROG (INTERVAL N.INT START N)
          (SETQ INTERVAL (GETP TABLE.NAME (QUOTE INTERVAL)))
          (SETQ N.INT (GETP TABLE.NAME (QUOTE N.INT)))
          (SETQ START (GETP TABLE.NAME (QUOTE START)))
          (SETQ N (ADD1 (IQUOTIENT (IDIFFERENCE TIME START)
                                   INTERVAL)))
          (COND
            ((GREATERP N N.INT)
              (RETURN (ST.COL.OUTER TABLE.NAME INTERVAL N.INT N TIME)))
            (T (RETURN (ST.COL.INNER TABLE.NAME INTERVAL N.INT N TIME])

(ST.COL.INNER
  [LAMBDA (TABLE.NAME INT NQUANT N TIME)
    (PROG (NROWS INDEX.ARRAY.PTR START.INDEX INDEX OLD.COL NEW.COL
                 START NEWEND ARRAY.END END.INDEX COUNT)
          (SETQ NROWS (GETP TABLE.NAME (QUOTE NROWS)))
          (SETQ INDEX.ARRAY.PTR (GETP TABLE.NAME (QUOTE INDEX.ARRAY.PTR)
                                      ))
          (SETQ START.INDEX (GETP TABLE.NAME (QUOTE START.INDEX)))
          (SETQ INDEX (IREMAINDER1 (PLUS (SUB1 N)
                                         START.INDEX)
                                   NQUANT))
          (SETQ OLD.COL (ELT INDEX.ARRAY.PTR INDEX))
          (COND
            ((EQ TIME (A.GETP (CDR OLD.COL)
                              (QUOTE START)))
              (RETURN OLD.COL)))
          [SETA INDEX.ARRAY.PTR INDEX
                (SETQ NEW.COL
                  (CONS (ARRAY NROWS)
                        (MAPCAR (CDR OLD.COL)
                                (FUNCTION (LAMBDA (BINDING)
                                    (COND
                                      ((EQ (CAR BINDING)
                                           (QUOTE BACKWARD))
                                        (CONS (QUOTE BACKWARD)
                                              OLD.COL))
                                      ((EQ (CAR BINDING)
                                           (QUOTE START))
                                        (CONS (QUOTE START)
                                              TIME))
                                      (T (CONS (CAR BINDING)
                                               (CDR BINDING]
          (A.PUT (CDR OLD.COL)
                 (QUOTE END)
                 TIME)
```

```
    (A.PUT (CDR (A.GETP (CDR OLD.COL)
                        (QUOTE FORWARD)))
           (QUOTE BACKWARD)
           NEW.COL)
    (A.PUT (CDR OLD.COL)
           (QUOTE FORWARD)
           NEW.COL)
    (SETQ NEWEND (A.GETP (CDR NEW.COL)
                         (QUOTE END)))
    (SETQ START (GETP TABLE.NAME (QUOTE START)))
    (SETQ ARRAY.END (IPLUS (ITIMES NQUANT INT)
                           START))
    [COND
      ((OR (NOT (NUMBERP NEWEND))
           (GREATERP NEWEND ARRAY.END))
        (SETQ END.INDEX (IREMAINDER1 (SUB1 START.INDEX)
                                     NQUANT)))
      (T (SETQ END.INDEX (IREMAINDER1 (IQUOTIENT (IDIFFERENCE
                                                   NEWEND START)
                                                 INT)
                                      NQUANT]
    (SETQ COUNT INDEX)
    (COND
      ([OR (NOT (NUMBERP NEWEND))
           (GREATERP NEWEND (IPLUS (GETP TABLE.NAME (QUOTE START))
                                   (ITIMES NQUANT INT]
        (PUT TABLE.NAME (QUOTE BEYOND.INDEX)
             NEW.COL)))
L   (SETQ COUNT (IREMAINDER1 COUNT NQUANT))
    (COND
      ((EQ COUNT END.INDEX)
        (SETA INDEX.ARRAY.PTR COUNT NEW.COL)
        (GO L1))
      (T (SETA INDEX.ARRAY.PTR COUNT NEW.COL)
         (SETQ COUNT (ADD1 COUNT))
         (GO L)))
L1  [RPTQ NROWS (SETA (CAR NEW.COL)
                      RPTN
                      (COPY (ELT (CAR OLD.COL)
                                 RPTN]
    (RETURN NEW.COL])
```

```
(ST.COL.OUTER
  [LAMBDA (TABLE.NAME INT NQUANT N TIME)
    (PROG (NROWS PTR OLDEND NEW.COL)
          (SETQ NROWS (GETP TABLE.NAME (QUOTE NROWS)))
          (SETQ PTR (GETP TABLE.NAME (QUOTE BEYOND.INDEX)))
      L   [COND
            ((EQ TIME (A.GETP PTR (QUOTE START)))
              (RETURN PTR))
            ((AND [NUMBERP (SETQ OLDEND (A.GETP PTR (QUOTE END]
                  (GREATERP TIME OLDEND))
              (SETQ PTR (A.GETP PTR (QUOTE FORWARD)))
              (GO L))
            ((EQ TIME OLDEND)
              (RETURN (A.GETP PTR (QUOTE FORWARD]
          [SETQ NEW.COL (CONS (ARRAY NROWS)
                              (MAPCAR (CDR PTR)
                                      (FUNCTION (LAMBDA (BINDING)
                                          (COND
                                            ((EQ (CAR BINDING)
                                                 (QUOTE BACKWARD))
                                              (CONS (QUOTE BACKWARD)
                                                    PTR))
                                            ((EQ (CAR BINDING)
                                                 (QUOTE START))
                                              (CONS (QUOTE START)
                                                    TIME))
                                            (T (CONS (CAR BINDING)
                                                     (CDR BINDING]
          (A.PUT (CDR PTR)
                 (QUOTE END)
                 TIME)
          (A.PUT (CDR (A.GETP (CDR PTR)
                              (QUOTE FORWARD)))
                 (QUOTE BACKWARD)
                 NEW.COL)
          (A.PUT (CDR PTR)
                 (QUOTE FORWARD)
                 NEW.COL)
          [RPTQ NROWS (SETA (CAR NEW.COL)
                            RPTN
                            (COPY (ELT (CAR PTR)
                                       RPTN]
          (COND
            ((GREATERP N (ADD1 NQUANT))
              (RETURN NEW.COL)))
          (PUT TABLE.NAME (QUOTE BEYOND.INDEX)
               NEW.COL)
          (RETURN NEW.COL])
```

```
(GET.COLS
  [LAMBDA (TABLE.NAME INIT END PRINT.SUP.FLG)
    (PROG (START PTR LST)
          (SETQ START (GETP TABLE.NAME (QUOTE START)))
          (COND
            [(AND (NUMBERP END)
                  (LESSP END START))
              (COND
                (PRINT.SUP.FLG (RETURN NIL))
                (T (PRIN1 "End time before start of table.")
                   (RETURN (CHARACTER 127]
            [(AND (NUMBERP END)
                  (LESSP END INIT))
              (COND
                (PRINT.SUP.FLG (RETURN NIL))
                (T (PRIN1 "End before initial time.")
                   (RETURN (CHARACTER 127]
            ((OR (NULL INIT)
                 (LESSP INIT START))
              (SETQ INIT START)))
          (SETQ PTR (GET.COL.PTR TABLE.NAME INIT))
          [SETQ LST (LIST (A.GETP PTR (QUOTE START]
      L   (COND
            ([NOT (NUMBERP (A.GETP PTR (QUOTE END]
              (RETURN (REVERSE LST)))
            ([AND (NUMBERP END)
                  (NOT (GREATERP END (A.GETP PTR (QUOTE END]
              (RETURN (REVERSE LST)))
            (T (SETQ PTR (A.GETP PTR (QUOTE FORWARD)))
               (SETQ LST (CONS (A.GETP PTR (QUOTE START))
                               LST))
               (GO L])

(GET.COL.PTR
  [LAMBDA (TABLE.NAME I.TIME)
    (PROG (T.START START.INDEX INDEX.ARRAY.PTR INTERVAL N.INT COL.PTR)
          (SETQ T.START (GETP TABLE.NAME (QUOTE START)))
          (COND
            ((AND I.TIME (NUMBERP I.TIME)
                  (LESSP I.TIME T.START))
              (SETQ I.TIME NIL)))
          (SETQ START.INDEX (GETP TABLE.NAME (QUOTE START.INDEX)))
          (SETQ INDEX.ARRAY.PTR (GETP TABLE.NAME
                                      (QUOTE INDEX.ARRAY.PTR)))
          [COND
            ((NULL I.TIME)
              (RETURN (ELT INDEX.ARRAY.PTR START.INDEX]
          (SETQ INTERVAL (GETP TABLE.NAME (QUOTE INTERVAL)))
          (SETQ N.INT (GETP TABLE.NAME (QUOTE N.INT)))
```

```
        [COND
          ((NOT (NUMBERP I.TIME)))
          [(NOT (ILESSP I.TIME (IPLUS T.START (ITIMES INTERVAL N.INT]
          (T (RETURN (ELT INDEX.ARRAY.PTR
                          (IREMAINDER1 (PLUS (IQUOTIENT (IDIFFERENCE
                                                          I.TIME
                                                          T.START)
                                                        INTERVAL)
                                                START.INDEX)
                                        N.INT]
        (SETQ COL.PTR (GETP TABLE.NAME (QUOTE BEYOND.INDEX)))
    L   (COND
          ([NOT (NUMBERP (A.GETP COL.PTR (QUOTE END]
            (RETURN COL.PTR))
          ([OR (NOT (NUMBERP I.TIME))
               (NOT (LESSP I.TIME (A.GETP COL.PTR (QUOTE END]
            (SETQ COL.PTR (A.GETP COL.PTR (QUOTE FORWARD)))
            (GO L))
          (T (RETURN COL.PTR])
```

E. Row Modification and Retrieval

The names actively used in a given table can be recovered with the function, NAMES, which returns a list of the names in order.

A row no longer needed because the named resource is no longer present can be removed by DELETE.ROW. If the set of named rows is not reduced to nothing, the function COPY.TABLE is used. This creates a new table, copying all the data from the old table except that in the row being deleted. The original name of the table then is reassigned to the new table. The old table is abandoned to be garbage-collected. A more efficient procedure could probably be devised. However, since it is not expected that rows will be deleted frequently, the efficiency of the procedure has not been considered critical, and the procedure given avoids any possible difficulty with the hash coding of the row names.

It is worth noting that COPY.TABLE can also be used directly to rearrange the order of the rows in a scroll table. The names of the rows need only be listed in the desired sequence and given to the function as the value of NAME.LIST. However, no check is made that the names in NAME.LIST are actually those used in the table, so that some care is needed in this use of COPY.TABLE.

The number of available rows in a table can be modified with EXPAND.TABLE. While the principal use of this function is to expand the number of available rows, it can also be used to reduce it, providing the new number is not less than that currently in use.

EXPAND.TABLE also acts by copying the table with the new value of NROWS, and then reassigning the origonal name to the copy. As with DELETE.ROW, the inefficiency of this procedure is not considered critical since it is not expected that EXPAND.Table will be used frequently.

These functions are defined as follows:

```
(NAMES
  [LAMBDA (TABLE.NAME)
    (PROG (NAMEARRAY NROWS (N 1)
                     NAMELIST)
          (SETQ NAMEARRAY (GETP TABLE.NAME (QUOTE ROWINDEX.TO.ROWNAME)))
          (SETQ NROWS (GETP TABLE.NAME (QUOTE NROWS)))
      L   [COND
            ((GREATERP N NROWS)
              (RETURN (REVERSE NAMELIST)))
            ((LISTP (ELT NAMEARRAY N))
              (SETQ NAMELIST (CONS (CAR (ELT NAMEARRAY N))
                                   NAMELIST]
          (SETQ N (ADD1 N))
          (GO L])

(DELETE.ROW
  [LAMBDA (TABLE NAME PRINT.SUPPRESS.FLG)
    (PROG (NAME.LIST)
          (COND
            ((SETQ NAME.LIST (NAMES TABLE)))
            ((NULL PRINT.SUP.FLG)
              (MAPPRINQ ("Do not recognize " TABLE "." TERPRI))
              (RETURN NIL))
            (T (RETURN NIL)))
          (COND
            ((MEMBER NAME NAME.LIST))
            ((NULL PRINT.SUP.FLG)
              (MAPPRINQ (NAME " not in use in " TABLE "." TERPRI))
              (RETURN NIL))
            (T (RETURN NIL)))
          (COND
            ((GREATERP (LENGTH NAME.LIST)
                       1)
              (COPY.TABLE TABLE (DREMOVE NAME NAME.LIST))
              (COND
                ((NULL PRINT.SUP.FLG)
                  (MAPRINT NAME.LIST T
                    "Table revised with the following row names:  "
                           (QUOTE %.)
                           ", ")))
              (RETURN (CHARACTER 127)))
```

```
                (T (MAKE.TABLE TABLE (GETP TABLE (QUOTE NROWS))
                               (GETP TABLE (QUOTE START))
                               (GETP TABLE (QUOTE N.INT))
                               (GETP TABLE (QUOTE INTERVAL))
                               (A.GETP (GETP TABLE (QUOTE DEFAULT.LIST))
                                       (QUOTE STATE.NAME))
                               T)
                   [COND
                     ((NULL PRINT.SUP.FLG)
                       (MAPPRINQ (TABLE " revised with no rows named."
                                        TERPRI]
                   (RETURN (CHARACTER 127])

(COPY.TABLE
  [LAMBDA (TABLE NAME.LIST)
    (PROG (DEFAULT.LIST ROWNAME.TO.ROWINDEX)
          (SETQ DEFAULT.LIST (GETP TABLE (QUOTE DEFAULT.LIST)))
          (MAKE.TABLE (QUOTE NEW.TABLE)
                      (GETP TABLE (QUOTE NROWS))
                      (GETP TABLE (QUOTE START))
                      (GETP TABLE (QUOTE N.INT))
                      (GETP TABLE (QUOTE INTERVAL))
                      (A.GETP DEFAULT.LIST (QUOTE STATE.NAME))
                      T)
          [MAPC NAME.LIST (FUNCTION (LAMBDA (X)
                    (NAME.ROW (QUOTE NEW.TABLE)
                              X T]
          (SETQ ROWNAME.TO.ROWINDEX (GETP (QUOTE NEW.TABLE)
                                           (QUOTE ROWNAME.TO.ROWINDEX)))
          [MAPC
            (GET.COLS TABLE)
            (FUNCTION (LAMBDA (X)
                (PROG (COL.PTR)
                      (SETQ COL.PTR (ST.COL (QUOTE NEW.TABLE)
                                            X))
                      (MAPC NAME.LIST
                            (FUNCTION (LAMBDA (Y)
                                (PROG (VAL)
                                      (COND
                                        ((EQ (SETQ VAL
                                                   (ST.ELT TABLE Y X))
                                             DEFAULT.LIST)
                                          (RETURN NIL))
                                        (T (SETA (CAR COL.PTR)
                                                 (GETHASH Y
                                                  ROWNAME.TO.ROWINDEX)
                                                 VAL]
          (SETPROPLIST TABLE (GETPROPLIST (QUOTE NEW.TABLE)))
          (RETURN NIL])
```

```
(EXPAND.TABLE
  [LAMBDA (TABLE NEW.NROWS)
    (PROG (COL.LIST NAMES ROWNAME.TO.ROWINDEX)
          (COND
            ((SETQ COL.LIST (GET.COLS TABLE)))
            (T (MAPPRINQ ("Do not recognize " TABLE "." TERPRI))
               (RETURN NIL)))
          (SETQ NAMES (NAMES TABLE))
          (COND
            ((LESSP NEW.NROWS (LENGTH NAMES))
              (MAPPRINQ ("Insufficient room for existing set of names."
                         TERPRI))
              (RETURN NIL)))
          (SETQ DEFAULT.LIST (GETP TABLE (QUOTE DEFAULT.LIST)))
          (MAKE.TABLE (QUOTE NEW.TABLE)
                      NEW.NROWS
                      (GETP TABLE (QUOTE START))
                      (GETP TABLE (QUOTE N.INT))
                      (GETP TABLE (QUOTE INTERVAL))
                      (A.GETP DEFAULT.LIST (QUOTE STATE.NAME))
                      T)
          [MAPC NAMES (FUNCTION (LAMBDA (X)
                    (NAME.ROW (QUOTE NEW.TABLE)
                              X T]
          (SETQ ROWNAME.TO.ROWINDEX (GETP (QUOTE NEW.TABLE)
                                           (QUOTE ROWNAME.TO.ROWINDEX)))
          [MAPC
            COL.LIST
            (FUNCTION (LAMBDA (X)
                (PROG (COL.PTR)
                      (SETQ COL.PTR (ST.COL (QUOTE NEW.TABLE)
                                            X))
                      (MAPC NAMES
                            (FUNCTION (LAMBDA (Y)
                                (PROG (VAL)
                                      (COND
                                        ((EQ (SETQ VAL
                                                   (ST.ELT TABLE Y X))
                                             DEFAULT.LIST)
                                          (RETURN NIL))
                                        (T (SETA (CAR COL.PTR)
                                                 (GETHASH Y
                                                   ROWNAME.TO.ROWINDEX)
                                                 VAL]
          (SETPROPLIST TABLE (GETPROPLIST (QUOTE NEW.TABLE)))
          (MAPPRINQ (TABLE " revised with " NEW.NROWS " rows." TERPRI))
          (RETURN (CHARACTER 127])
```

VIII PRIMARY DATA FUNCTIONS

Given that a scroll table exists and the desired names have been given its rows, the next step is to be able to enter and recover data from the cells of the table. The nature of the data and its format is not of concern yet, only the actual entry and retrieval operations. Also, no consideration has been given, yet, to the specification of consistency within a table, or to the construction of the applicable models. Therefore, the functions of concern here are not the top level ones, but those used by the top level functions to actually enter or access the data in the table. The data entry functions also fire any demons that may be attached, and so must include the procedures that check the preconditions of any demons and fire them if required.

A. Data Entry

The actual entry of data into the table is done with the function TABLE.SETA. The data is required to be formatted completely before it is called, and is given to it as the variable A.LIST. TABLE.SETA uses a subsidiary function, TABLE.SETA.1, that makes the entry into the individual cells that are affected. This is done so that the element demons that may be attached to the data in those cells can be transferred to the new data. TABLE.SETA.1 therefore operates on a single column. TABLE.SETA uses MAPC to execute TABLE.SETA.1 on each column time found by GET.COLS for the required interval.

For reasons that are discussed later, there is need also for a version that will enter the A-list only in those columns for which the existing entry has a specified state name. This is accomplished with TABLE.SETA.TEST.

Note that this function has no guards against failure. Neither does it check to see if any new columns are needed. As discussed later, it is used under conditions where these checks are not needed. It does, however, fire the row and table demons; the element and column demons are fired by TABLE.SETA.1 which it uses.

In TABLE.SETA.1, after an entry has been made to a cell, any element and column demons are fired by FIRE.DEMON.LIST. The same function is used by TABLE.SETA and TABLE.SETA.TEST after all entries have been made to fire the row and table demons. The detailed structure of a demon is specified later. Here, it is sufficient to observe that the first clause in the COND of FIRE.DEMON.LIST tests the precondition of a demon. If this clause is true, or non-NIL, the second clause fires the demon. FIRE.DEMON.LIST executes this test, and, if satisfied, fires the demon for all demons in DEMON.LIST.

There is also need for a function that can accept a list of intervals, and apply TABLE.SETA in each interval. The function that does this is TABLE.SETA.LST. The list is required to be a list of dotted pairs. For each pair, the first element is the start of an interval, the second its end.

TABLE.SETA.TEST.LST is a similar function that uses TABLE.SETA.TEST. That is, it takes a list of intervals expressed as dotted pairs. Within each interval, TABLE.SETA.TEST makes the entry if, and only if, the old state name is that specified.

These functions are defined as follows:

```
(TABLE.SETA
  [LAMBDA (TABLE ROW.NAME INIT END A.LIST)
    (PROG (I.TIME ROWINDEX E.TIME DEMON.LIST)
          (ST.SCROLL TABLE)
          [COND
            ((OR (NULL INIT)
                 (LESSP INIT S.CLOCK))
              (SETQ I.TIME (NEXTLOWER TABLE S.CLOCK)))
            (T (SETQ I.TIME (NEXTLOWER TABLE INIT]
          (COND
            [(SETQ ROWINDEX (GETHASH ROW.NAME
                                     (GETP TABLE
                                           (QUOTE ROWNAME.TO.ROWINDEX]
            (T (MAPPRINQ ("Either " TABLE " or " ROW.NAME
                                     " not recognized." TERPRI))
               (RETURN NIL)))
          (COND
            ((AND END (NUMBERP END)
                  (GREATERP END I.TIME))
              (SETQ E.TIME (NEXTHIGHER TABLE END)))
            ((AND END (NUMBERP END))
              (PRIN1 "End too early.  No entry.")
              (TERPRI)
              (RETURN NIL))
            (T (SETQ E.TIME NIL)))
          (ST.COL TABLE I.TIME)
          (COND
            (E.TIME (ST.COL TABLE E.TIME)))
          [MAPC (GET.COLS TABLE I.TIME E.TIME)
                (FUNCTION (LAMBDA (X)
                    (TABLE.SETA.1 TABLE ROWINDEX X (COPY A.LIST]
          (COND
            ((SETQ DEMON.LIST (A.GETP (ELT (GETP TABLE
                                                 (QUOTE
                                                  ROWINDEX.TO.ROWNAME))
                                           ROWINDEX)
                                      (QUOTE DEMONS)))
              (FIRE.DEMON.LIST DEMON.LIST A.LIST NIL)))
```

```
            (COND
              ((SETQ DEMON.LIST (A.GETP (GETP TABLE (QUOTE A.LIST))
                                        (QUOTE DEMONS)))
                (FIRE.DEMON.LIST DEMON.LIST A.LIST NIL)))
            (RETURN T])

(TABLE.SETA.1
  [LAMBDA (TABLE ROWINDEX TIME A.LIST)
    (PROG (COL.PTR OLD.LIST DEMON.LIST)
          (SETQ COL.PTR (GET.COL.PTR TABLE TIME))
          (SETQ OLD.LIST (ELT (CAR COL.PTR)
                              ROWINDEX))
          (SETQ DEMON.LIST (A.GETP OLD.LIST (QUOTE DEMONS)))
          (COND
            (DEMON.LIST (A.PUT A.LIST (QUOTE DEMONS)
                               DEMON.LIST)))
          (SETA (CAR COL.PTR)
                ROWINDEX A.LIST)
          (COND
            (DEMON.LIST (FIRE.DEMON.LIST DEMON.LIST A.LIST OLD.LIST)))
          (COND
            ((SETQ DEMON.LIST (A.GETP COL.PTR (QUOTE DEMONS)))
              (FIRE.DEMON.LIST DEMON.LIST A.LIST OLD.LIST])

(TABLE.SETA.TEST
  [LAMBDA (TABLE NAME I.TIME E.TIME STATE A.LIST)
    (PROG (ROWINDEX DEMON.LIST)
          (ST.SCROLL TABLE)
          [SETQ ROWINDEX (GETHASH NAME (GETP TABLE
                                             (QUOTE ROWNAME.TO.ROWINDEX]
          (COND
            ((NOT (NUMBERP E.TIME))
              (SETQ E.TIME NIL)))
          [MAPC (GET.COLS TABLE I.TIME E.TIME)
                (FUNCTION (LAMBDA (X)
                    (COND
                      ((EQUAL STATE (A.GETP (ST.ELT TABLE NAME X)
                                            (QUOTE STATE-NAME)))
                        (TABLE.SETA.1 TABLE ROWINDEX X (COPY A.LIST]
          (COND
            ((SETQ DEMON.LIST
                   (A.GETP (ELT (GETP TABLE (QUOTE ROWINDEX.TO.ROWNAME))
                                ROWINDEX)
                           (QUOTE DEMONS)))
              (FIRE.DEMON.LIST DEMON.LIST A.LIST NIL)))
          (COND
            ((SETQ DEMON.LIST (A.GETP (GETP TABLE (QUOTE A.LIST))
                                      (QUOTE DEMONS)))
              (FIRE.DEMON.LIST DEMON.LIST A.LIST NIL])
```

```
(FIRE.DEMON.LIST
  [LAMBDA (DEMON.LIST A.LIST OLD.LIST)
    (MAPC DEMON.LIST (FUNCTION (LAMBDA (X)
              (COND
                ((APPLY* (3RD X)
                         (CAR X)
                         (2ND X)
                         A.LIST)
                  (APPLY* (4TH X)
                          OLD.LIST A.LIST (5TH X])

(TABLE.SETA.LST
  [LAMBDA (TABLE ROW.NAME LST A.LIST)
    (MAPC LST (FUNCTION (LAMBDA (X)
              (TABLE.SETA TABLE ROW.NAME (CAR X)
                          (CDR X)
                          A.LIST])

(TABLE.SETA.TEST.LST
  [LAMBDA (TABLE NAME LST STATE A.LIST)
    (MAPC LST (FUNCTION (LAMBDA (X)
              (TABLE.SETA.TEST (CAR X)
                               (CDR X)
                               STATE A.LIST])
```

B. Data Retrieval and Display

The primary function used for the recovery of data from a cell of the scroll table is ST.ELT. TIME, here, does not need to be the start time of a column; it will find the column whose interval contains the desired time. Note also that ST.ELT returns the entire content of the cell, including any demons. Hence the value returned by ST.ELT can be difficult to read. There is need of functions that can extract the required information from it and present it in a convenient form.

There are two functions used for printing out the table. PRINT.TABLE prints out the virtual table; i.e. as if all columns had been made. PRINT.PROP prints out on the columns that have actually been created, and so gives a condensed version of the table. Both functions allow for a property to be named. If not named, it is assumed to be STATE.NAME. What is printed for each cell is the result of applying A.GETP with the indicated property name to the value returned by ST.ELT.

Table 1, given before, was printed out by the call of PRINT.TABLE and Table 2 by the call of PRINT.PROP. The pseudo-natural language interface shown in Figure 1 translates the input command into a call on one of these functions.

PRINT.PROP prints, at most, six columns which provides a convenient display on the paper. PRINT.TABLE can print any number of columns, but does so in blocks of six columns at most. For this purpose, it uses PRINT.TABLE.1 to print each block.

These functions print an asterisk if, in a given cell, the designated property is null or if the property is unspecified, and so assumed to be STATE.NAME, and the value is the default state name of the table. This enhances the visibility of the entries that are likely to be the more significant ones.

The actual entry in the cell of a table, as retrieved by ST.ELT, can be quite long and involved. This is particularly true when demons are involved, although the detailed structure of demons are not discussed until later. As an example, Table 6A is an illustration of what is retrieved by ST.ELT. The printout in Table 6B shows the same data, reformatted by the special print function, PRINT.ENTRY. The significance of the various components in the entry printed in Table 6, and the reasons for the duplications apparent there, are discussed later. PRINT.ENTRY provides a printout of the contents of a cell that is much easier to understand than is the direct printout of the value of ST.ELT.

Table 6
DATA IN A TABLE CELL

Table 6A
DIRECT PRINTOUT

```
(ST.ELT 'PILOTS 'BAKER 120]
((ENTRY . MISSION)(STATE-NAME . ASG)(LEVEL . 5)(START . 35)
(END . 195)(MISSION-ID . M1)(FPC . A4)(DEMONS (NIL NIL DUMMY
WATCH.1 ((PILOTS BAKER 35 195 DUMMY WATCH.1 NIL NIL) NIL
MISSION M1))))
```

TABLE 6B
READABLE PRINTOUT

```
(PRINT.ENTRY 'PILOTS 'BAKER 120]
  ENTRY:  MISSION
  STATE-NAME:  ASG
  LEVEL:  5
  START:  35
  END:  195
  MISSION-ID:  M1
  FPC:  A4
  Demons:
       Demon:
            NIL
            NIL
            DUMMY
            WATCH.1
            Arg list:
                 (PILOTS BAKER 35 195 DUMMY WATCH.1 NIL NIL)
                 NIL
                 MISSION
                 M1
```

The functions described here are defined as follows:

```
(ST.ELT
  [LAMBDA (TABLE NAME TIME PRINT.SUP.FLG)
    (PROG (ROWINDEX COL.PTR)
          [COND
            [(SETQ ROWINDEX (GETHASH NAME (GETP TABLE
                                               (QUOTE ROWNAME.TO.ROWINDEX]
            (PRINT.SUP.FLG (RETURN NIL))
            (T (MAPPRINQ ("No information on " NAME "." TERPRI))
               (RETURN (CHARACTER 127]
          [COND
            ((LESSP TIME (GETP TABLE (QUOTE START)))
              (COND
                (PRINT.SUP.FLG (RETURN NIL))
                (T (PRIN1 "Time too early.  No information.")
                   (RETURN (CHARACTER 127]
          (SETQ COL.PTR (GET.COL.PTR TABLE (NEXTLOWER TABLE TIME)))
          (RETURN (ELT (CAR COL.PTR)
                       ROWINDEX])
```

```
(PRINT.PROP
  [LAMBDA (TABLE INIT END PROP)
    (PROG (COL.LIST DEFAULT FLG K)
          (COND
            ((OR (NULL INIT)
                 (LESSP INIT S.CLOCK))
              (SETQ INIT S.CLOCK)))
          (COND
            ((NULL PROP)
              (SETQQ PROP STATE.NAME)
              (SETQ FLG T)))
          (SETQ COL.LIST (GET.COLS TABLE INIT END))
          [COND
            ((GREATERP (LENGTH COL.LIST)
                       6)
              (SETQ COL.LIST (LDIFF COL.LIST (NTH COL.LIST 7]
          [SETQ DEFAULT (CDAR (GETP TABLE (QUOTE DEFAULT.LIST]
          (MAPPRINQ ((RPTQ 2 (TERPRI))
                     (TAB 3)
                     "Table:  " TABLE TERPRI "Property:  " PROP))
          [COND
            (FLG (MAPPRINQ ((TAB 30)
                            "(" DEFAULT " replaced by *.)")))
            (T (MAPPRINQ ((TAB 30)
                          "(Non-occurence of " PROP " indicated by *.)"]
          (RPTQ 2 (TERPRI))
          (PRIN1 "Name\Time")
          (SETQ K 15)
          [MAPC (COPY COL.LIST)
                (FUNCTION (LAMBDA (X)
                    (PROG (NIL)
                          (TAB K)
                          (PRIN1 (CONVERT.TIME X))
                          (SETQ K (IPLUS K 10]
          (TERPRI)
```

```
            [MAPC (NAMES TABLE)
                  (FUNCTION (LAMBDA (X)
                      (PROG (NIL)
                            (PRIN1 X)
                            (SETQ K 15)
                            [MAPC (COPY COL.LIST)
                                  (FUNCTION (LAMBDA (Y)
                                      (PROG (NIL)
                                            (SETQ VAL
                                              (A.GETP (ST.ELT TABLE X Y)
                                                      PROP))
                                            (TAB K)
                                            (SETQ K (IPLUS K 10))
                                            (COND
                                              ((AND FLG (EQ VAL DEFAULT))
                                                (PRIN1 "*"))
                                              (VAL (PRIN1 VAL))
                                              (T (PRIN1 "*"]
                            (TERPRI]
            (TERPRI)
            (RETURN (CHARACTER 127])

(PRINT.TABLE
  [LAMBDA (TABLE INIT END PROP)
    (PROG (INTERVAL FLG DEFAULT NAMES TRIAL.END)
          [COND
            ([OR (NULL INIT)
                 (LESSP INIT (GETP TABLE (QUOTE START]
              (SETQ INIT (GETP TABLE (QUOTE START]
          (COND
            [(SETQ INTERVAL (GETP TABLE (QUOTE INTERVAL]
            (T (MAPPRINQ ("Do not recognize " TABLE "." TERPRI))
               (RETURN NIL)))
          [COND
            ((OR (NULL END)
                 (NOT (NUMBERP END)))
              (SETQ END (IPLUS INIT (ITIMES 6 INTERVAL]
          (COND
            ((NULL PROP)
              (SETQQ PROP STATE.NAME)
              (SETQ FLG T)))
          [SETQ DEFAULT (CDAR (GETP TABLE (QUOTE DEFAULT.LIST]
          (MAPPRINQ ((RPTQ 2 (TERPRI))
                     (TAB 3)
                     "Table:  " TABLE TERPRI "Property:  " PROP))
          [COND
            (FLG (MAPPRINQ ((TAB 30)
                            "(" DEFAULT " replaced by *)")))
            (T (MAPPRINQ ((TAB 30)
                          "(Non-occurence of " PROP " indicated by *)"]
          (RPTQ 2 (TERPRI))
```

```
          (SETQ NAMES (NAMES TABLE))
      L   (SETQ TRIAL.END (IPLUS INIT (ITIMES 6 INTERVAL)))
          (COND
            ((LESSP END TRIAL.END)
              (SETQ TRIAL.END END)))
          (PRINT.TABLE.1 NAMES INIT TRIAL.END END INTERVAL DEFAULT PROP
                         FLG)
          (COND
            ((LESSP TRIAL.END END)
              (SETQ INIT TRIAL.END)
              (GO L))
            (T (RETURN (CHARACTER 127])

(PRINT.TABLE.1
  [LAMBDA (NAMES INIT TRIAL.END END INTERVAL DEFAULT PROP FLG)
    (PROG (NIL)
          (PRIN1 "Name\Time")
          (SETQ K 15)
          (SETQ COL.TIME INIT)
      L   (TAB K)
          (PRIN1 (CONVERT.TIME COL.TIME))
          (SETQ COL.TIME (IPLUS COL.TIME INTERVAL))
          (COND
            ((LESSP COL.TIME TRIAL.END)
              (SETQ K (IPLUS K 10))
              (GO L)))
          (TERPRI)
          [MAPC NAMES (FUNCTION (LAMBDA (X)
                    (PROG (NIL)
                          (SETQ K 15)
                          (SETQ COL.TIME INIT)
                          (PRIN1 X)
                      L1  (TAB K)
                          (SETQ VAL (A.GETP (ST.ELT TABLE X COL.TIME)
                                            PROP))
                          (COND
                            ((AND FLG (EQ VAL DEFAULT))
                              (PRIN1 "*"))
                            (VAL (PRIN1 VAL))
                            (T (PRIN1 "*")))
                          (SETQ COL.TIME (IPLUS COL.TIME INTERVAL))
                          (SETQ K (IPLUS K 10))
                          (COND
                            ((LESSP COL.TIME TRIAL.END)
                              (GO L1))
                            (T (TERPRI)
                               (RETURN NIL]
          (RPTQ 2 (TERPRI])
```

```
(PRINT.ENTRY
  [LAMBDA (TABLE NAME TIME)
    (PROG (A.LIST)
          (SETQ A.LIST (ST.ELT TABLE NAME TIME))
          [MAPC
            A.LIST
            (FUNCTION (LAMBDA (X)
                (COND
                  ((NOT (EQUAL (CAR X) (QUOTE DEMONS)))
                    (MAPPRINQ ((TAB 10) (CAR X) ":  " (CDR X) TERPRI)))
                  (T
                    (TAB 10)
                    (PRIN1 "Demons:  ")
                    (MAPC
                      (CDR X)
                      (FUNCTION (LAMBDA (Y)
                          (PROG (NIL)
                                (TAB 15)
                                (PRIN1 "Demon:  ")
                                (MAPC Y
                                  (FUNCTION (LAMBDA (Z)
                                     (COND
                                       ((ATOM Z)
                                         (MAPPRINQ (TERPRI (TAB 20) Z)))
                                       ((LISTP Z)
                                         (MAPPRINQ ((TAB 20)
                                                    "Arg list:  "
                                                    TERPRI))
                                         (MAPC Z (FUNCTION (LAMBDA (U)
                                                   (MAPPRINQ ((TAB 25)
                                                     U TERPRI]
          (RETURN (CHARACTER 127])
```

IX MODEL FUNCTIONS

The functions described in this section are those used for creating and manipulating the resource model used by a scheduler. The set of models used by a scheduler are encoded as an a-list with the value of MODELS which is used as a property name in the A.LIST of the table. That is, a given model can be recovered by

(A.GETP (A.GETP (GETP <table name> 'A.LIST) 'MODELS) <model name>)

Note that, by a model, we mean the set of rules and constraints that describe a particular kind of entry to the data of a scheduler, such as assignment to a mission, rest after assignment, and so on. The resource model of a scheduler is the collection of models that have been defined for the scheduler. The entire resource model is the value of the property MODELS.

A. Creation and Display of a Model

A new model for an existing scheduler, or scroll table, is entered using the following functions. The entire function has been divided into three functions for convenience in making changes and additions, should this be necessary later. MAKE.RES.MODEL is the top level function. It handles those properties that are necessary in any model, specifically STATE-NAME and LEVEL, and makes sure that these are given acceptable values. It then calls BUILD.MODEL, which covers certain optional properties. Finally, ENTER.E.LIST is called, which constructs the E-list (for entry list) of the model. Additional properties can be added as a modification of the model, as discussed later.

ENTER.E.LIST obtains the type of the model and the names of the functions used for and by the watch and set demons, if any. Three types have been defined, so far. Type 1 is an entry, such as MISSION in Table 4, that must be entered in its entirety or not at all. Type 2 is made from a specified start time as far as possible or until a specified duration is obtained. MISSION-REST, in Table 4, is an example. Type 3 permits discontinuous entry from a given start time until the required duration has been obtained. TRAINING, in Table 4, is an example. Other types could be defined, but these have been found sufficient so far.

Were a new type to be defined, ENTER.E.LIST is the only one of the model building functions that would require modification. If a new property-value pair required, either BUILD.MODEL could be modified to include it, or its addition handled as a modification of the model.

The function, MAKE.RES.MODEL ends with the user option of seeing a display of the model, as in Table 3. Note that this is accomplished with PRINT.LIST, defined before. No new print function is needed. On the other hand, the entire process model, including all the models that have been defined for a scheduler, can be printed with PRINT.MODELS. Table 4 was printed with this function.

Finally, GET.R.MODEL.PROP has been defined as a convenient function for obtaining the value of a specific property, at any depth, in MODELS. Its variable, PROP.LIST, is a list of names, starting with the model name, that are property names in the A-lists at successive levels in the model. It returns the value of the last property in the list.

Table 7 illustrates the dialog mode that is used by the model creation functions. It shows the creation of MISSION as a model in the scheduler named PILOTS.

Table 7
CREATION OF A MODEL

```
(MAKE.RES.MODEL 'PILOTS 'MISSION]

For the entry type MISSION, what state name?  ASG

What level?  5

Is the duration of MISSION specified?
Enter its value if so, else enter ]:  ]

Does MISSION have an ID associated with it?
Enter its name if so, else enter ]:  MISSION-ID

Does MISSION have a class code associated with it?
Enter its name if so, else enter ]:  FPC

Does MISSION have a successor entry type associated with it?
Enter its name if so, else enter ]:  MISSION-REST

What type is MISSION?
Enter 1, 2 or 3, or ? if want definitions:  ?


Type 1:  Continuous entry over entire specified interval.
Type 2:  Continuous entry from start as far as possible until
                                                          end.
Type 3:  Discontinuous entry allowed.  From start until duration
                                                       obtained.

What type is MISSION?
Enter 1, 2 or 3, or ? if want definitions:  1

What function is used as the watch demon?
(Enter ] if none.  Else function name.)  WATCH.1

What function is called by WATCH.1?
(Enter ] if none.  Else name.)  DIALOG.1


Do you want to see the model?  (Y or ])  Y
```

The entry type named MISSION in the scheduler named PILOTS is as follows:

```
STATE-NAME:  ASG.
LEVEL:  5.
ID-LABEL:  MISSION-ID.
CLASS-LABEL:  FPC.
POST-ENTRY:  MISSION-REST.
E-LIST:
    TYPE:  1.
    WATCH-DEMON:
        WATCH.1:
            CALL-FUNCTION:  DIALOG.1.
```

The result of the exercise shown in Table 7 can be seen if (GETP 'PILOTS 'A.LIST) is used. The table A-list is returned as:

```
((DEMONS) (MODELS (MISSION (STATE-NAME . ASG) (LEVEL . 5)
(ID-LABEL . ID-MISSION) (CLASS-LABEL . FPC) (POST-ENTRY . MISSION-REST)
(E-LIST (TYPE . 1) (WATCH-DEMON WATCH.1 (CALL-FUNCTION . DIALOG.1))))))).
```

Alternatively, if (GET.R.MODEL.PROP 'PILOTS '(MISSION)) is called to recover only the MISSION model, it is returned as:

```
((STATE-NAME . ASG) (LEVEL . 5) (ID-LABEL . ID-MISSION)
(CLASS-LABEL . FPC) (POST-ENTRY . MISSION-REST)
(E-LIST (TYPE . 1) (WATCH-DEMON WATCH.1 (CALL-FUNCTION . DIALOG.1)))).
```

These several functions are:

```
(MAKE.RES.MODEL
  [LAMBDA (TABLE ENTRY.TYPE)
    (PROG (STATE.NAME LEVEL MODEL MODEL.LIST)
          (COND
            ((NULL TABLE)
              (MAPPRINQ ("No table name specified." TERPRI))
              (RETURN NIL))
            [(SETQ A.LIST (GETP TABLE (QUOTE A.LIST]
            (T (MAPPRINQ (TABLE
                         " not recognized as the name of a table."
                            TERPRI))
               (RETURN NIL)))
```

```
L   (COND
      ((NULL ENTRY.TYPE)
        (PRIN1 "What is the name of the entry type?  ")
        (SETQ ENTRY.TYPE (READ))
        (GO L))
      ((A.GETP# A.LIST (LIST (QUOTE MODEL)
                             ENTRY.TYPE))
        (MAPPRINQ (ENTRY.TYPE
                   " already used as the name of an entry type."
                             TERPRI
           "If intention was to alter it, use ALTER.RES.MODEL.)"
                             TERPRI))
        (RETURN NIL)))
    (RPTQ 2 (TERPRI))
L1  (MAPPRINQ ("For the entry type " ENTRY.TYPE
                                    ", what state name?  "))
    (COND
      ((SETQ STATE.NAME (READ)))
      (T (PRIN1 "A state name is required.")
         (TERPRI)
         (GO L1)))
    (COND
      ((NOT (ATOM STATE.NAME))
        (PRIN1 "State name must be an atom.")
        (TERPRI)
        (GO L1))
      ((GREATERP (NCHARS STATE.NAME)
                 7)
        (MAPPRINQ (STATE.NAME
                        " is too long.  Limit it to 7 characters."
                          TERPRI))
        (GO L1)))
L2  (TERPRI)
    (PRIN1 "What level?  ")
    (SETQ LEVEL (READ))
    (COND
      ((NOT (NUMBERP LEVEL))
        (PRIN1 "Level must be a number.  ")
        (TERPRI)
        (GO L2))
      ((OR (LESSP LEVEL 0)
           (GREATERP LEVEL 10))
        (PRIN1 "Level must be non-negative and not more than 10.")
        (TERPRI)
        (GO L2)))
    (TERPRI)
    (SETQ MODEL (LIST (CONS (QUOTE STATE-NAME)
                            STATE.NAME)
                      (CONS (QUOTE LEVEL)
                            LEVEL)))
```

```
          (BUILD.MODEL ENTRY.TYPE MODEL)
          [COND
            ((SETQ MODEL.LIST (A.GETP A.LIST (QUOTE MODELS)))
              (A.PUT MODEL.LIST ENTRY.TYPE MODEL))
            (T (A.PUT A.LIST (QUOTE MODELS)
                      (LIST (CONS ENTRY.TYPE MODEL]
          (RPTQ 2 (TERPRI))
          (PRIN1 "Do you want to see the model?  (Y or ])  ")
          (COND
            ((EQ (READ)
                 (QUOTE Y))
              [MAPPRINQ (TERPRI "The entry type named " ENTRY.TYPE
                                " in the scheduler named " TABLE
                                " is as follows:" (RPTQ 2 (TERPRI]
              (PRINT.LIST MODEL 10))
            (T (PRIN1 "OK")))
          (RETURN (CHARACTER 127])

(BUILD.MODEL
  [LAMBDA (ENTRY.TYPE MODEL)
    (PROG (DURATION ID.LABEL CLASS.LABEL POST.ENTRY)
          (MAPPRINQ ("Is the duration of " ENTRY.TYPE " specified?"
                                           TERPRI
                                "Enter its value if so, else enter ]:  "))
      L   (COND
            ((SETQ DURATION (READ))
              (COND
                ((NOT (NUMBERP DURATION))
                  (PRIN1
             "The duration must be a number.  Re-enter the duration:  ")
                  (GO L)))
              (A.PUT MODEL (QUOTE DURATION)
                     DURATION)))
          (MAPPRINQ (TERPRI "Does " ENTRY.TYPE
                            " have an ID associated with it?" TERPRI
                            "Enter its name if so, else enter ]:  "))
          (COND
            ((SETQ ID.LABEL (READ))
              (A.PUT MODEL (QUOTE ID-LABEL)
                     ID.LABEL)))
          (MAPPRINQ (TERPRI "Does " ENTRY.TYPE
                            " have a class code associated with it?"
                            TERPRI
                            "Enter its name if so, else enter ]:  "))
          (COND
            ((SETQ CLASS.LABEL (READ))
              (A.PUT MODEL (QUOTE CLASS-LABEL)
                     CLASS.LABEL)))
          (MAPPRINQ (TERPRI "Does " ENTRY.TYPE
                    " have a successor entry type associated with it?"
                            TERPRI
                            "Enter its name if so, else enter ]:  "))
```

```
            (COND
              ((SETQ POST.ENTRY (READ))
                (A.PUT MODEL (QUOTE POST-ENTRY)
                       POST.ENTRY)))
            (A.PUT MODEL (QUOTE E-LIST)
                   (ENTER.E.LIST ENTRY.TYPE])

(ENTER.E.LIST
  [LAMBDA (ENTRY.TYPE)
    (PROG (TYPE E.LIST WATCH CALL SET)
      L   (MAPPRINQ (TERPRI "What type is " ENTRY.TYPE "?" TERPRI
                            "Enter 1, 2 or 3, or ? if want definitions:  "))
          (COND
            ((EQ (SETQ TYPE (READ))
                 (QUOTE ?))
              (GO DESC))
            ((OR (NOT (FIXP TYPE))
                 (LESSP TYPE 1)
                 (GREATERP TYPE 3))
              (MAPPRINQ ("Type must be an integer between 1 and 3."
                         TERPRI))
              (GO L)))
          (SETQ E.LIST (LIST (CONS (QUOTE TYPE)
                                   TYPE)))
          (MAPPRINQ (TERPRI "What function is used as the watch demon?"
                            TERPRI
                            "Enter its name if any, else enter ]:  "))
          [COND
            ((SETQ WATCH (READ))
              (A.ADDPROP E.LIST (QUOTE WATCH-DEMON)
                         WATCH)
              (MAPPRINQ (TERPRI "What function is called by " WATCH
                                "?" TERPRI
                                "(Enter ] if none.  Else name.)  "))
              (COND
                ((SETQ CALL (READ))
                  (A.PUT# E.LIST (LIST (QUOTE WATCH-DEMON)
                                       (QUOTE CALL-FUNCTION))
                          CALL]
          (COND
            ((EQ TYPE 1)
              (RETURN E.LIST)))
          (MAPPRINQ (TERPRI "What function is used as the set demon?"
                            TERPRI
                            "Enter its name if any, else enter ]:  "))
```

```
        [COND
          ((SETQ SET (READ))
            (A.ADDPROP E.LIST (QUOTE SET-DEMON)
                       SET)
            (MAPPRINQ (TERPRI "What function is called by " SET "?"
                              TERPRI
                              "(Enter ] if none.  Else name.)  "))
            (COND
              ((SETQ CALL (READ))
                (A.PUT# E.LIST (LIST (QUOTE SET-DEMON)
                                     (QUOTE CALL-FUNCTION))
                       CALL]
          (RETURN E.LIST)
      DESC(MAPPRINQ ((RPTQ 2 (TERPRI))

            "Type 1:  Continuous entry over entire specified interval."
                    TERPRI
    "Type 2:  Continuous entry from start as far as possible until end."
                    TERPRI
"Type 3:  Discontinuous entry allowed.  From start until duration
                                                          obtained."
                    TERPRI))
          (GO L])

(PRINT.MODELS
  [LAMBDA (TABLE)
    (PROG (NIL)
          (TERPRI)
          [MAPPRINQ ("The following models have been entered into "
                     TABLE ":" (RPTQ 2 (TERPRI]
          (PRINT.LIST (A.GETP (GETP TABLE (QUOTE A.LIST))
                              (QUOTE MODELS))
                      10)
          (RPTQ 2 (TERPRI))
          (RETURN (CHARACTER 127])

(GET.R.MODEL.PROP
  [LAMBDA (TABLE PROP.LIST)
    (A.GETP# (GETP TABLE (QUOTE A.LIST))
             (CONS (QUOTE MODELS)
                   PROP.LIST])
```

B. Modification of a Model

Once the model exists, it can be modified by ALTER.RES.MODEL. The call of this function requires only the identification of the table or scheduler, and of the model. It then asks for a list of the properties to be changed or added. The properties that it can understand are those at the top level of the model. For example, if one wishes to change the value of some property in the E-list, the propery name E-LIST is given to it. When the function reaches that property name, it will call ENTER.E.LIST so that the entire E-list will be reconstructed.

As an example, Table 8 shows the result of modifying the model developed in Table 7. Two changes are desired. The value of LEVEL is being changed to 4, and a new property called CONDITION is to be entered with the value TEST.FN. (This property has no significance in ACS.1. It is used here only as an illustration.)

Table 8
MODIFYING A RESOURCE MODEL

```
(ALTER.RES.MODEL 'PILOTS 'MISSION]

Do you want to see the model as it stands?
(Enter ] if no.)  ]

Are you satisfied with the model?
(Enter ] if no.  Otherwise will exit.)  ]

What properties do you wish to change or add?
Enter as a list, terminated with ].  LEVEL CONDITION]

     What is the value of CONDITION?  TEST.FN
     What is the value of LEVEL?  4

Do you want to see the model as it stands?
(Enter ] if no.)  Y
          STATE-NAME:  ASG.
          LEVEL:  4.
          ID-LABEL:  ID-MISSION.
          CLASS-LABEL:  FPC.
          POST-ENTRY:  MISSION-REST.
          E-LIST:
              TYPE:  1.
              WATCH-DEMON:
                  WATCH-FN:
                      CALL-FUNCTION:  DIALOG-FN.
          CONDITION:  TEST.FN.

Are you satisfied with the model?
(Enter ] if no.  Otherwise will exit.)  Y
OK
```

The function does not terminate until the user declares himself satisfied. If, on the last question of Table 8, the answer was] or NIL, the function would have restarted on the third question.

This function can reduce the values of any of its properties to NIL but cannot remove the process model in its entirety. Removal is done by REM.R.MODEL. REM.R.MODEL actually leaves the model name in MODELS, but with NIL value. Its presence does not prevent a later use of MAKE.RES.MODEL to redine the model.

With these functions, models can be created, modified and deleted at will, once the scroll table has been created. The functions are:

```
(ALTER.RES.MODEL
  [LAMBDA (TABLE ENTRY.TYPE)
    (PROG (A.LIST MODEL LST)
      L   (COND
            ((NULL TABLE)
              (MAPPRINQ ("No table name given." TERPRI))
              (RETURN NIL))
            ([NULL (SETQ A.LIST (GETP TABLE (QUOTE A.LIST]
              (MAPPRINQ ("Do not recognize " TABLE " as a table name."
                                             TERPRI))
              (RETURN NIL))
            ((NULL ENTRY.TYPE)
              (PRIN1 "What entry type?  ")
              (SETQ ENTRY.TYPE (READ))
              (GO L))
            [(SETQ MODEL (A.GETP# A.LIST (LIST (QUOTE MODELS)
                                               ENTRY.TYPE]
            (T (MAPPRINQ (ENTRY.TYPE
                          " not currently a resource model in "
                                TABLE "." TERPRI
                          "If a new model, use MAKE.RES.MODEL."
                                TERPRI))
               (RETURN NIL)))
      L1  (MAPPRINQ (TERPRI
                     "Do you want to see the model as it stands?"
                          TERPRI "(Enter ] if no.)  "))
          (COND
            ((READ)
              (PRINT.LIST MODEL 10)))
          (MAPPRINQ (TERPRI "Are you satisfied with the model?" TERPRI
                            "(Enter ] if no.  Otherwise will exit.)  "))
          [COND
            ((READ)
              (PRIN1 "OK")
              (RETURN (CHARACTER 127]
          (MAPPRINQ (TERPRI
                     "What properties do you wish to change or add?"
                     TERPRI "Enter as a list, terminated with ].  "))
      L2  (COND
            ((SETQ X (READ))
              (SETQ LST (CONS X LST))
              (GO L2)))
          [MAPC LST (FUNCTION (LAMBDA (Y)
                    (COND
                      ((EQ Y (QUOTE E-LIST))
                        (A.PUT MODEL (QUOTE E-LIST)
                               (ENTER.E.LIST ENTRY.TYPE)))
                      (T (MAPPRINQ ((TAB 5)
                                    "What is the value of " Y "?  "))
                         (A.PUT MODEL Y (READ]
          (SETQ LST NIL)
          (GO L1])
```

```
(REM.R.MODEL
  [LAMBDA (TABLE ENTRY.TYPE)
    (PROG (A.LIST)
          (COND
            ((NULL TABLE)
              (MAPPRINQ ("No table name." TERPRI))
              (RETURN NIL))
            ([NULL (SETQ A.LIST (GETP TABLE (QUOTE A.LIST]
              (MAPPRINQ ("Do not recognize " TABLE " as a table name."
                                             TERPRI))
              (RETURN NIL))
            ((NULL ENTRY.TYPE)
              (MAPPRINQ ("No entry type specified." TERPRI))
              (RETURN NIL))
            ((NULL (A.GETP# A.LIST (LIST (QUOTE MODELS)
                                         ENTRY.TYPE)))
              (MAPPRINQ (ENTRY.TYPE " not a model currently in " TABLE
                                    "." TERPRI))
              (RETURN NIL)))
          [COND
            ((GREATERP (LENGTH (A.GETP A.LIST (QUOTE MODELS)))
                       1)
              (A.REMPROP (A.GETP A.LIST (QUOTE MODELS))
                         ENTRY.TYPE))
            (T (A.REMPROP A.LIST (QUOTE MODELS]
          (MAPPRINQ (ENTRY.TYPE " deleted as a model in " TABLE "."
                                TERPRI))
          (RETURN (CHARACTER 127])
```

X DEMON MANIPULATION

In this section, we consider the functions that manipulate the demons that may be required to maintain the retroactive consistency of the table, or for other purposes.

The form of a demon in ACS.1 is specified to be a list of the form:

(<property> <value> <function.1> <function.2> <arg.list>)

The first three terms are the precondition of the demon. TABLE.SETA, and its subordinate functions, listed above, first evaluates (<function.1> (A.GET A.LIST <property>) <value>) where A.LIST is the A-list being entered by TABLE.SETA. For example, if the property is LEVEL and function.1 is LESSP, the demon will be fired if the value of LEVEL in the new entry is below the specified value.

If the precondition is true, or not NIL, function.2 is applied in the form (<function.2> <old-a-list> <new-a-list> <arg-list>). The old a-list is the a-list returned by ST.ELT prior to making any entry. The third term, or the second argument, is the a-list being entered. The final argument, arg-list, can be used to carry along any required information from the original entry process that set up the demon. The call of function.2 is the firing of the demon.

The discussion, so far, has concentrated on the demons that are used to maintain retroactive consistency. These demons are always element demons, they are set on the data in the cells of the table. Demons that are used to initiate alert messages, or to initiate more general system actions, may be set on elements, columns, rows or on the table as a whole. For example, a demon set on a column will be checked whenever the data in any cell in that column is changed. A row demon is checked whenever the data in any cell in the given row is changed. A table demon is fired whenever the data in any cell of the table is changed. The element and column demons are checked by TABLE.SETA.1 and TABLE.SETA.2 since these functions are accessing the columns and cells. The row and table demons are checked by TABLE.SETA itself after the entry has been made. Note that the row and table demons are not permitted to use the old a-list since there may have been entries to many cells. The same format is used for the list that is the demon, but, in the call of function.2, its first argument, which is the old a-list, is automatically set to NIL.

A. Attachment of Demons

The principal function for the creation and attachment of a demon is SET.DEMON. If both NAME and I.TIME are null, the demon is attached to the table as a whole, in the table A-list. If Name is given, but I.TIME is NIL, it is attached as a row demon. If NAME is NIL, but I.TIME is given, it is attached as a column demon. Otherwise, it is attached as an element demon. The actual entries are made by SET.TABLE.DEMON, SET.ROW.DEMON, SET.COL.DEMON or SET.EL.DEMON, as appropriate.

Note that these functions create the list that is the demon in the standard form as the demon is entered as one of the listed values of the property DEMONS in the required location or locations.

Note that these functions cannot be embedded directly in TABLE.SETA that enters data into the table. The arguments of the functions, including, for example, the demon functions themselves, are specified in the model. Indeed, it is the model that specifies if demons are required to enforce retroactive consistency. Hence these functions are called, not by TABLE.SETA, but by higher level functions that use the model.

The functions are:

```
(SET.DEMON
  [LAMBDA (TABLE NAME I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST)
    (PROG (START)
          (COND
            ((NULL TABLE)
              (PRIN1 "No table.  ")
              (RETURN NIL))
            [(SETQ START (GETP TABLE (QUOTE START]
            (T (PRIN1 "Do not recognize table.  ")
               (RETURN NIL)))
          (COND
            ((OR (NULL FN1)
                 (NULL FN2))
              (PRIN1 "Functions not both given.  ")
              (RETURN NIL)))
          (COND
            ((AND E.TIME (OR (NULL I.TIME)
                             (LESSP I.TIME START)))
              (SETQ I.TIME START))
            ((AND I.TIME (LESSP I.TIME START))
              (SETQ I.TIME START)))
          [COND
            ((AND (NOT (NUMBERP E.TIME))
                  (OR E.TIME I.TIME))
              (SETQ E.TIME (QUOTE INDEF]
```

```
          (COND
            ((AND (NULL NAME)
                  (NULL I.TIME))
              (SET.TABLE.DEMON TABLE FN1 FN2 PROP VAL ARG.LIST))
            ((NULL I.TIME)
              (SET.ROW.DEMON TABLE NAME FN1 FN2 PROP VAL ARG.LIST))
            ((NULL NAME)
              (SET.COL.DEMON TABLE I.TIME E.TIME FN1 FN2 PROP VAL
                             ARG.LIST))
            (T (SET.EL.DEMON TABLE NAME I.TIME E.TIME FN1 FN2 PROP VAL
                             ARG.LIST])

(SET.EL.DEMON
  [LAMBDA (TABLE NAME I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST)
    (PROG (DEMON)
          (SETQ DEMON (LIST PROP VAL FN1 FN2 ARG.LIST))
          (MAPC (GET.COLS TABLE I.TIME E.TIME)
                (FUNCTION (LAMBDA (X)
                    (A.ADDPROP (ST.ELT TABLE NAME X)
                               (QUOTE DEMONS)
                               DEMON])

(SET.COL.DEMON
  [LAMBDA (TABLE I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST)
    (PROG (NIL)
          (ST.COL TABLE I.TIME)
          (COND
            ((NUMBERP E.TIME)
              (ST.COL TABLE E.TIME))
            (T (SETQ E.TIME NIL)))
          (MAPC (GET.COLS TABLE I.TIME E.TIME)
                (FUNCTION (LAMBDA (X)
                    (A.ADDPROP (GET.COL.PTR TABLE X)
                               (QUOTE DEMONS)
                               (LIST PROP VAL FN1 FN2 ARG.LIST])

(SET.ROW.DEMON
  [LAMBDA (TABLE NAME FN1 FN2 PROP VAL ARG.LIST)
    (A.ADDPROP [ELT (GETP TABLE (QUOTE ROWINDEX.TO.ROWNAME))
                    (GETHASH NAME (GETP TABLE (QUOTE
                                               ROWNAME.TO.ROWINDEX]
               (QUOTE DEMONS)
               (LIST PROP VAL FN1 FN2 ARG.LIST])

(SET.TABLE.DEMON
  [LAMBDA (TABLE FN1 FN2 PROP VAL ARG.LIST)
    (A.ADDPROP (GETP TABLE (QUOTE A.LIST))
               (QUOTE DEMONS)
               (LIST PROP VAL FN1 FN2 ARG.LIST])
```

B. Removal of Demons

There is also a need for functions to remove demons at will. They are removed by KILL.DEMON which, like SET.DEMON, uses the presence or absence of NAME and I.TIME to sort out whether it is an element, column, row or table demon that is to be killed, and calls the appropriate subordinate function.

There is also a function, KILL.DEMON.1, that accepts (instead of I.TIME and E.TIME) a list of dotted pairs, each of which indicates an interval of time and applies KILL.DEMON to each interval. This function can be applied to either element or column demons.

All of these functions act by constructing an exact duplicate of the demon, and then removing any matching demon from the list of demons. Any error in the reconstruction of the demon will cause the process to fail. Therefore, all key information must be available at the time the demon is to be removed. One way to be certain that this information will be available is to include it in the argument list that is part of the demon. This may lead to redundant storage of some information that may be present in the data to which the demon is attached, but this redundancy seems a small price to pay.

These functions are:

```
(KILL.DEMON
  [LAMBDA (TABLE NAME I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST)
    (COND
      ((AND (NULL NAME)
            (NULL I.TIME)
            (NULL E.TIME))
        (KILL.TABLE.DEMON TABLE FN1 FN2 PROP VAL ARG.LIST))
      ((AND (NULL I.TIME)
            (NULL E.TIME))
        (KILL.ROW.DEMON TABLE NAME FN1 FN2 PROP VAL ARG.LIST))
      ((OR (NULL I.TIME)
           (NULL E.TIME))
        (PRIN1 "Confusion.  ")
        NIL)
      ((NULL NAME)
        (KILL.COL.DEMON TABLE I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST))
      (T (KILL.EL.DEMON TABLE NAME I.TIME E.TIME FN1 FN2 PROP VAL
                        ARG.LIST])
```

```
(KILL.EL.DEMON
  [LAMBDA (TABLE NAME I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST)
    (PROG (DEMON)
          (SETQ DEMON (LIST PROP VAL FN1 FN2 ARG.LIST))
          (MAPC (GET.COLS TABLE I.TIME E.TIME)
                (FUNCTION (LAMBDA (X)
                    (PROG (CELL DEMON.LIST)
                          (SETQ CELL (ST.ELT TABLE NAME X))
                          (SETQ DEMON.LIST (A.GETP CELL (QUOTE DEMONS)))
                          (COND
                            ((NULL DEMON.LIST)
                              (RETURN NIL))
                            ((EQUAL DEMON.LIST (LIST DEMON))
                              (A.PUT CELL (QUOTE DEMONS)
                                     NIL))
                            (T (DREMOVE DEMON DEMON.LIST])

(KILL.COL.DEMON
  [LAMBDA (TABLE I.TIME E.TIME FN1 FN2 PROP VAL ARG.LIST)
    (PROG (DEMON)
          (SETQ DEMON (LIST PROP VAL FN1 FN2 ARG.LIST))
          (MAPC (GET.COLS TABLE I.TIME E.TIME)
                (FUNCTION (LAMBDA (X)
                    (PROG (DEMON.LIST COL.PTR)
                          (SETQ COL.PTR (GET.COL.PTR TABLE X))
                          (SETQ DEMON.LIST (A.GETP COL.PTR
                                                   (QUOTE DEMONS)))
                          (COND
                            ((NULL DEMON.LIST)
                              (RETURN NIL))
                            ((EQUAL DEMON.LIST (LIST DEMON))
                              (A.PUT COL.PTR (QUOTE DEMONS)
                                     NIL))
                            (T (DREMOVE DEMON DEMON.LIST])

(KILL.ROW.DEMON
  [LAMBDA (TABLE NAME FN1 FN2 PROP VAL ARG.LIST)
    (PROG (NAME.ELT DEMON)
          [SETQ NAME.ELT (ELT (GETP TABLE (QUOTE ROWINDEX.TO.ROWNAME))
                              (GETHASH NAME (GETP TABLE (QUOTE
                                                   ROWNAME.TO.ROWINDEX]
          (SETQ DEMON (LIST PROP VAL FN1 FN2 ARG.LIST))
          (COND
            ((EQUAL (A.GETP NAME.ELT (QUOTE DEMONS))
                    (LIST DEMON))
              (A.PUT NAME.ELT (QUOTE DEMONS)
                     NIL))
            (T (DREMOVE DEMON (A.GETP NAME.ELT (QUOTE DEMONS])
```

```
(KILL.TABLE.DEMON
  [LAMBDA (TABLE FN1 FN2 PROP VAL ARG.LIST)
    (PROG (DEMON.LIST DEMON)
          (SETQ DEMON.LIST (A.GETP (GETP TABLE (QUOTE A.LIST))
                                   (QUOTE DEMONS)))
          (SETQ DEMON (LIST PROP VAL FN1 FN2 ARG.LIST))
          (COND
            ((NULL DEMON.LIST))
            ((EQUAL DEMON.LIST (LIST DEMON))
              (A.PUT (GETP TABLE (QUOTE A.LIST))
                     (QUOTE DEMONS)
                     NIL))
            (T (DREMOVE DEMON DEMON.LIST])

(KILL.DEMON.1
  [LAMBDA (TABLE NAME LST FN1 FN2 PROP VAL ARG.LIST)
    (MAPC LST (FUNCTION (LAMBDA (X)
              (KILL.DEMON TABLE NAME (CAR X)
                          (CDR X)
                          FN1 FN2 PROP VAL ARG.LIST]
```

C. Self-Destruct Property

The watch and set demons, in particular, require that their first action should be the removal of all occurences of themselves. Their intended actions involve the data to which the watch demon is attached, or the locations at which the set demons are attached. If the demons are not removed, they are likely to be refired as a result of the action of the demon itself. This will at least lead to the possibility of a considerable amount of unnecessary work, and may lead to thrashing. As a general principle, if the primary purpose of a demon is to modify the data to which it is attached, its first action should be the removal of all occurences of itself. The demon can then be reattached after the data has been modified, if appropriate.

Further, the watch and set demons, where the model specifies both, form a complementary pair. When either is fired, its first action should be to eliminate not only all occurences of itself, but also all occurences of its complement.

As indicated in the previous section, it is convenient to include in the argument list all the information necessary to ensure that the demon can be removed. This principle is carried one step further. We specify that the first element in the arg.list of a demon shall be a list. This list is required to list, in order, the first eight variables that should be given to KILL.DEMON. The final variable of KILL.DEMON is the argument list itself. Hence the demon can be killed by:

```
(APPLY 'KILL.DEMON (APPEND (CAR ARG.LIST) (LIST ARG.LIST))).
```

The function APPLY is used here since the variables, as a result of the APPEND operation, are contained in a list.

In order to kill both members of a complimentary pair when either is fired, we adopt another convention. If a single demon is involved, we specify that the second element in its argument list shall be NIL. If there is a second demon, this element shall be T. In the latter case, the third element shall also be a list to which KILL.DEMON can be applied after appending the argument list.

This convention could be extended to include any number of demons in a set. However, we have not found occasion to use sets that are larger than a complementary pair.

The functions that exploit this structure of a demon are AUTO.KILL.DEMON and AUTO.KILL.DEMON.1. The latter uses KILL.DEMON.1, which uses a list of dotted pairs each of which identifies an interval of time. Both functions take only the argument list as the variable. Both return the tail of the argument list starting after the part that is used for the kill demon operation, and which may contain information that is useful for other purposes.

As an example, consider the printout of a complete entry into one cell of a table shown in Table 6. In the readable format of Table 6B, the entry after "Demons:" is the list of demons. There is only one demon present The argument list of this demon is shown. The first element is indeed a list, and lists the first eight of the variables that must be given to KILL.DEMON to remove all occurences. The ninth argument of KILL.DEMON is the entire argument list. The second element of the argument list is NIL, there is no complementary demon that should also be killed. When AUTO.KILL.DEMON is executed with this argument list, (MISSION M1) is returned, the tail of the argument list.

A related function, KILL.DEMON.FN, has also been defined. This function examines the list of demons attached to a specified cell of the table. If the function.2 of a demon matches the specified demon function, AUTO.KILL.DEMON or AUTO.KILL.DEMON.1 is executed on it. This function is used when a previous entry is to be changed as a result of an externally originated command; i.e., not as a result of firing a demon. For example, the entry may be cancelled. It is necessary to first remove any watch and set demons that may have been set on the data, before actually removing the data. Otherwise the demons might be fired. The procedure for doing this is to find one cell that contains the data, or that might have a set demon attached, and then using KILL.DEMON.FN on the contents of this cell.

These functions are defined as follows:

```
(AUTO.KILL.DEMON
  [LAMBDA (ARG.LIST)
    (PROG (NIL)
          (APPLY (QUOTE KILL.DEMON)
                 (APPEND (CAR ARG.LIST)
                         (LIST ARG.LIST)))
          (COND
            ((2ND ARG.LIST)
              (APPLY (QUOTE KILL.DEMON)
                     (APPEND (3RD ARG.LIST)
                             (LIST ARG.LIST)))
              (RETURN (NTH ARG.LIST 4)))
            (T (RETURN (NTH ARG.LIST 3])

(AUTO.KILL.DEMON.1
  [LAMBDA (ARG.LIST)
    (PROG (NIL)
          (APPLY (QUOTE KILL.DEMON.1)
                 (APPEND (CAR ARG.LIST)
                         (LIST ARG.LIST)))
          (COND
            ((2ND ARG.LIST)
              (APPLY (QUOTE KILL.DEMON.1)
                     (APPEND (3RD ARG.LIST)
                             (LIST ARG.LIST)))
              (RETURN (NTH ARG.LIST 4)))
            (T (RETURN (NTH ARG.LIST 3])

(KILL.DEMON.FN
  [LAMBDA (TABLE NAME TIME DEMON.FN TYPE)
    (PROG (DEMON.LIST)
          (COND
            ((SETQ DEMON.LIST (A.GETP (ST.ELT TABLE NAME TIME)
                                      (QUOTE DEMONS)))
              (MAPC DEMON.LIST (FUNCTION (LAMBDA (X)
                        (COND
                          [(AND (EQ TYPE 3)
                                (EQUAL (4TH X)
                                       DEMON.FN))
                            (AUTO.KILL.DEMON.1 (5TH X]
                          ((EQUAL (4TH X)
                                  DEMON.FN)
                            (AUTO.KILL.DEMON (5TH X])
```

XI TOP LEVEL FUNCTIONS

We come now to the functions that coordinate these separate actions and features, and that provide top level access to the scheduler.

A. Top Level Data Entry

The function used for the actual entry of data in accordance with the rules of the models is ENTER.TYPE. Among its other variables, it takes two time variables, TIME.1 and TIME.2. TIME.1 is the start of the entry. If the entry is of type 1, TIME.2 is its end. Otherwise, TIME.2 is the duration. Note that the duration can also be specified in the model. However, if TIME.2 is specified on the call of the function, the model is not checked. TIME.2 will then govern. If TIME.2 is not specified, and the model does not contain a duration, the function asks for a value.

OVERRIDE is a flag that appears in several functions. It specifies whether or not the entry is to be permitted when the existing data has the same level as the desired entry. As a top level function, OVERRIDE is usually NIL, and the entry is only permitted providing its level is greater than that of the existing data. However, these functions are also used by the watch and set demons. As discussed, these demons have the effect of generating a delayed call of the entry function when they are fired. In this case, we want the entry to be able to override the data that was entered originally, so that the data in the cells are brought up to date. When this is required, the demons call these functions with the OVERRIDE flag set to T.

The principal task of ENTER.TYPE is to sort out what type the entry is, as defined in the model for the named entry type. In addition, it checks the condition of the table and establishes whether or not the entry is permitted and, if so, how. It does this by calling CHECK.ENTRY.1, CHECK.ENTRY.2 or CHECK.ENTRY.3, depending on the type specified by the model.

CHECK.ENTRY.1 is used for type 1 entries. These must be entered in their entirety or not at all. It returns T if the specified level is greater than the level in any cell into which the data will be placed, or, if OVERRIDE is set, not less than. Otherwise, it returns NIL, which will then cause ENTER.TYPE to refuse the entry.

CHECK.ENTRY.2 is used for type 2 for which the entry is to be made as far as possible from the start, but not beyond the end. The possibility of entrance is, again, determined by the level in the

existing data in the table. It returns the start time of the first column encountered in which the data has too high a value of LEVEL. If none is encountered, it returns T. If the value returned is T, the entrance will be made from the value of INIT to the value of END. Otherwise, it will be made from the value of INIT to the value returned by CHECK.ENTRY.2, providing the latter is greater than INIT. Note that it is quite possible that no data will be entered for a type 2 entry. However, it may still cause demons to be set, as is discussed later.

CHECK.ENTRY.3 is for type 3, in which the entry may be made discontinuously. Therefore CHECK.ENTRY.3 returns a list of two lists. The first list is one of intervals, expressed as dotted pairs, in which the desired entry can be made. The total time covered by these intervals will add to the required duration. The second list is one of intervals, also expressed as dotted pairs, in which the entry is forbidden because of the presence of data with too high a value of LEVEL. It is a list of the gaps in the first list and is used to determine where set demons should be attached. Two functions are used by CHECK.ENTRY.3. LIMIT.LEVEL.FWD determines how far forward from a given starting time the given entry can be made as determined by its level. ADMIT.LEVEL.FWD determines the first time after the specified time at which the given entry can be made.

Once the suitability of the entry has been checked and, if appropriate, its parameters determined, ENTER.TYPE calls ENTER.TYPE.1, ENTER.TYPE.2, or ENTER.TYPE.3, again depending on the type specified by the model. These functions actually make the entry and set whatever demons are required.

A detail should be noted that can be overlooked easily. Before ENTER.TYPE.2 is called by ENTER.TYPE, a column is created by the call of ST.COL just after the time returned by CHECK.ENTRY.2. This column spans just one interval of the table. The reason for this is so the set demon can be attached to the cell in this column, and will not be replicated by any later manipulations of the table, or called on any entry except one in that particular column.

Note that these functions construct the argument list of the demons so as to permit the auto-kill operation, implementing the self-destruct property.

The A-lists used by the ENTER.TYPE functions, and passed to TABLE.SETA, are constructed by MAKE.ALIST or MAKE.ALIST.1. The latter is used by a type 3 entry, and includes the lists of dotted pairs returned by CHECK.ENTRY.3 as the values of ENTRIES and GAPS.

The MAKE.ENTRY functions also call PUT.INSTANCE if, but only if, the model specifies an ID-LABEL. This function adds a property-value pair to a list that is stored in the table A-list, where there is a list of the value of INSTANCES. This list is composed of a set of lists, each of which is the value of a property, whose name is the name of an entry type. It is the appropriate one of these lists that PUT.INSTANCE adds the property-value pair whose property is the ID of the entry. If the entry type is type 1 or 2, the value of ID in this list is (<row name> <start> <end>) where start and end are the start and end times of the actual entry. If it is type 3, the value is (<row name> <list.1> <list.2>) where list.1 is a list of dotted pairs indicating intervals over which the entry was made, and list.2 the complementary list of gaps in the entry. The value entered by PUT.INSTANCE permits accessing the entry by its ID, without having to search the table.

A further detail should be noted. The first time PUT.INSTANCE adds a property-value pair under a given entry type, it actually constructs (<entry type> DEFAULT (ID <list>)). The inclusion of DEFAULT avoids ever needing to remove all the values of the entry type, so that A.REMPROP, and its related functions, can remove all the significant property-value pairs in the structure.

The function, ENTER.TYPE, also uses LAST.ENTRY if the entry is type 3 to determine the time of completion. It searches a list of dotted pairs, returning T if the second member of any pair is non-numeric, otherwise returning the largest of the second members.

The function NLESSP should be mentioned. It is the same as LESSP except that, if either value is not a number, it returns T, rather than generating an error message.

These functions permit entry into the scroll table of data in accordance with the rules and constraints of the resource model. They also set the demons that will continue to enforce those rules and constraints.

The functions described here are defined as follows:

```
(ENTER.TYPE
  [LAMBDA (TABLE NAME TIME.1 TIME.2 ENTRY.TYPE ID CLASS PRINT.SUP.FLG
                 OVERRIDE)
    (PROG (MODEL TYPE END E.TIME DURATION ENTRY.LIST POST.ENTRY)
          (COND
            [(SETQ MODEL (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE]
            ((NULL PRINT.SUP.FLG)
              (MAPPRINQ (ENTRY.TYPE " is not a model in " TABLE "."
                                    TERPRI))
              (RETURN NIL))
            (T (RETURN NIL)))
```

```
[SETQ TYPE (A.GETP# MODEL (LIST (QUOTE E-LIST)
                                (QUOTE TYPE]
[COND
  ((EQ TYPE 1)
    (SETQ END TIME.2))
  ((NUMBERP TIME.2)
    (SETQ END (IPLUS TIME.1 TIME.2]
[COND
  [(NOT (NUMBERP TIME.1))
    (COND
      (PRINT.SUP.FLG (RETURN NIL))
      (T (PRIN1 "No start.")
         (RETURN (CHARACTER 127]
  ((LESSP TIME.1 (GETP TABLE (QUOTE START)))
    (COND
      [PRINT.SUP.FLG (SETQ TIME.1 (GETP TABLE (QUOTE START]
      (T (PRIN1 "Start too early.")
         (RETURN (CHARACTER 127]
[COND
  ((AND (NUMBERP END)
        (LESSP END TIME.1))
    (COND
      (PRINT.SUP.FLG (RETURN NIL))
      (T (PRIN1 "End too early.")
         (RETURN (CHARACTER 127]
(COND
  [(A.GETP MODEL (QUOTE ID-LABEL))
    (COND
      ((NULL ID)
        (PRIN1 "What ID?  ")
        (SETQ ID (READ]
  (T (SETQ ID NIL)))
(COND
  ((AND (EQ TYPE 1)
        (CHECK.ENTRY.1 TABLE NAME TIME.1 TIME.2
                       (A.GETP MODEL (QUOTE LEVEL))
                       OVERRIDE))
    (ENTER.TYPE.1 TABLE NAME TIME.1 TIME.2 ENTRY.TYPE ID
                  CLASS)
    (SETQ E.TIME (NEXTHIGHER TABLE TIME.2)))
```

```
                ((EQ TYPE 2)
                  [COND
                    ((NUMBERP TIME.2)
                      (SETQ DURATION TIME.2))
                    [(SETQ DURATION (A.GETP MODEL (QUOTE DURATION]
                    (T (PRIN1 "What duration?  ")
                       (SETQ DURATION (READ]
                  (SETQ E.TIME (CHECK.ENTRY.2 TABLE NAME TIME.1
                                              (IPLUS TIME.1 DURATION)
                                              (A.GETP MODEL (QUOTE LEVEL))
                                              OVERRIDE))
                  [COND
                    ((OR (NOT (NUMBERP E.TIME))
                         (GREATERP E.TIME (IPLUS TIME.1 DURATION)))
                      (SETQ E.TIME (IPLUS TIME.1 DURATION]
                  (ENTER.TYPE.2 TABLE NAME TIME.1 (IPLUS TIME.1 DURATION)
                                E.TIME ENTRY.TYPE ID CLASS))
                 [(EQ TYPE 3)
                  [COND
                    ((NUMBERP TIME.2)
                      (SETQ DURATION TIME.2))
                    [(SETQ DURATION (A.GETP MODEL (QUOTE DURATION]
                    (T (PRIN1 "What duration?  ")
                       (SETQ DURATION (READ]
                  (SETQ ENTRY.LIST (CHECK.ENTRY.3 TABLE NAME TIME.1 DURATION
                                                  (A.GETP MODEL
                                                          (QUOTE LEVEL))
                                                  OVERRIDE))
                  (ENTER.TYPE.3 TABLE NAME TIME.1 DURATION ENTRY.LIST
                                ENTRY.TYPE ID CLASS)
                  (SETQ E.TIME (LAST.ENTRY (CAR ENTRY.LIST]
                 (T (PRIN1 "Can't.")
                    (TERPRI)
                    (RETURN NIL)))
                (COND
                  ([AND (NUMBERP E.TIME)
                        (SETQ POST.ENTRY (A.GETP MODEL (QUOTE POST-ENTRY]
                    (ENTER.TYPE TABLE NAME E.TIME NIL POST.ENTRY ID CLASS T
                                OVERRIDE)))
                (COND
                  ((NULL PRINT.SUP.FLG)
                    (PRIN1 "OK")))
                (RETURN (CHARACTER 127])

(CHECK.ENTRY.1
  [LAMBDA (TABLE NAME INIT END LEVEL)
    (EVERY (GET.COLS TABLE INIT END)
           (FUNCTION (LAMBDA (X)
               (NOT (LESSP LEVEL (A.GETP (ST.ELT TABLE NAME X)
                                         (QUOTE LEVEL])
```

```
(CHECK.ENTRY.2
  [LAMBDA (TABLE NAME INIT END LEVEL)
    (PROG (NIL)
          [MAPC (GET.COLS TABLE INIT END)
                (FUNCTION (LAMBDA (X)
                    (COND
                      ((GREATERP (A.GETP (ST.ELT TABLE NAME X)
                                         (QUOTE LEVEL))
                                 LEVEL)
                        (RETURN X]
          (RETURN T])

(CHECK.ENTRY.3
  [LAMBDA (TABLE NAME INIT DURATION LEVEL)
    (PROG (TRIAL NEW.TRIAL LST.1 LST.2)
          (SETQ TRIAL INIT)
      L   (COND
            [(SETQ NEW.TRIAL (LIMIT.LEVEL.FWD TABLE NAME TRIAL LEVEL))
              (COND
                ((NOT (NUMBERP NEW.TRIAL))
                  (SETQ LST.1 (CONS (CONS TRIAL (IPLUS TRIAL DURATION))
                                    LST.1))
                  (RETURN (LIST LST.1 LST.2)))
                [(NOT (GREATERP (IPLUS TRIAL DURATION)
                                NEW.TRIAL))
                  (SETQ LST.1 (CONS (CONS TRIAL (IPLUS TRIAL DURATION))
                                    (RETURN LST.1 LST.2]
                (T (SETQ LST.1 (CONS (CONS TRIAL NEW.TRIAL)
                                     LST.1))
                   (SETQ DURATION (IDIFFERENCE (IPLUS TRIAL DURATION)
                                               NEW.TRIAL))
                   (SETQ TRIAL NEW.TRIAL)
                   (GO L]
            ((SETQ NEW.TRIAL (LIMIT.LEVEL.FWD TABLE NAME TRIAL LEVEL T))
              (COND
                ((NOT (NUMBERP NEW.TRIAL))
                  (SETQ LST.2 (CONS (CONS TRIAL (QUOTE INDEF))
                                    LST.2))
                  (RETURN (LIST LST.1 LST.2)))
                (T (SETQ LST.2 (CONS (CONS TRIAL NEW.TRIAL)
                                     LST.2))
                   (SETQ TRIAL NEW.TRIAL)
                   (GO L])
```

```
(LIMIT.LEVEL.FWD
  [LAMBDA (TABLE NAME START LEVEL OVERRIDE)
    (PROG (PTR)
          (COND
            ((GREATERP (A.GETP (ST.ELT TABLE NAME START)
                               (QUOTE LEVEL))
                       LEVEL)
              (RETURN NIL))
            ((AND (NULL OVERRIDE)
                  (EQP (A.GETP (ST.ELT TABLE NAME START)
                               (QUOTE LEVEL))
                       LEVEL))
              (RETURN NIL)))
          (SETQ PTR (GET.COL.PTR TABLE START))
       L  [COND
            [(GREATERP (A.GETP (ST.ELT TABLE NAME START)
                               (QUOTE LEVEL))
                       LEVEL)
              (RETURN (A.GETP PTR (QUOTE START]
            ((AND (NULL OVERRIDE)
                  (EQP (A.GETP (ST.ELT TABLE NAME START)
                               (QUOTE LEVEL))
                       LEVEL))
              (RETURN (A.GETP PTR (QUOTE START]
          (COND
            ([NOT (NUMBERP (SETQ START (A.GETP PTR (QUOTE END]
              (RETURN T)))
          (SETQ PTR (A.GETP PTR (QUOTE FORWARD)))
          (GO L])

(ADMIT.LEVEL.FWD
  [LAMBDA (TABLE NAME START LEVEL OVERRIDE)
    (PROG (PTR)
      L   (SETQ PTR (GET.COL.PTR TABLE START))
          (COND
            ((LESSP (A.GETP (ST.ELT TABLE NAME START)
                            (QUOTE LEVEL))
                    LEVEL)
              (RETURN START))
            ((AND OVERRIDE (EQP (A.GETP (ST.ELT TABLE NAME START)
                                        (QUOTE LEVEL))
                                LEVEL))
              (RETURN START)))
          (COND
            ([NOT (NUMBERP (SETQ START (A.GETP PTR (QUOTE END]
              (RETURN NIL)))
          (GO L])
```

AD-A046 312 STANFORD RESEARCH INST MENLO PARK CALIF F/G 12/2
THE SCHEDULERS OF ACS.1.(U)
SEP 77 M C PEASE N00014-77-C-0308
UNCLASSIFIED SRI-TR-14 NL

2 OF 2
AD
A046 312

END
DATE
FILMED
12-77
DDC

```
(ENTER.TYPE.1
  [LAMBDA (TABLE NAME INIT END ENTRY.TYPE ID CLASS)
    (PROG (A.LIST ID.LABEL CLASS.LABEL WATCH.FN)
          (SETQ A.LIST (MAKE.ALIST TABLE NAME INIT END ENTRY.TYPE
                                   ID CLASS))
          [COND
            (ID (PUT.INSTANCE TABLE ENTRY.TYPE ID (LIST NAME INIT END]
          (TABLE.SETA TABLE NAME INIT END A.LIST)
          (COND
            ([SETQ WATCH.FN (GET.R.MODEL.PROP TABLE
                                              (LIST ENTRY.TYPE
                                                    (QUOTE E-LIST)
                                                    (QUOTE WATCH-DEMON]
              (SET.EL.DEMON TABLE NAME INIT END (QUOTE DUMMY)
                            (CAR WATCH.FN)
                            NIL NIL (LIST (LIST TABLE NAME INIT END
                                                (QUOTE DUMMY)
                                                (CAR WATCH.FN)
                                                NIL NIL)
                                          NIL ENTRY.TYPE ID])

(ENTER.TYPE.2
  [LAMBDA (TABLE NAME INIT END E.TIME ENTRY.TYPE ID CLASS)
    (PROG (A.LIST ARG.LIST.1 ARG.LIST.2 WATCH.FN SET.FN)
          (SETQ A.LIST (MAKE.ALIST TABLE NAME INIT END ENTRY.TYPE ID
                                   CLASS))
          [SETQ LEVEL (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE
                                                    (QUOTE LEVEL]
          [COND
            ((GREATERP E.TIME INIT)
              (TABLE.SETA TABLE NAME INIT E.TIME A.LIST)
              (SETQ WATCH.FN (GET.R.MODEL.PROP TABLE
                                               (LIST ENTRY.TYPE
                                                     (QUOTE E-LIST)
                                                     (QUOTE WATCH-DEMON]
          [COND
            ((LESSP E.TIME END)
              [SETQ SET.FN (GET.R.MODEL.PROP TABLE
                                             (LIST ENTRY.TYPE
                                                   (QUOTE E-LIST)
                                                   (QUOTE SET-DEMON]
              (ST.COL TABLE (IPLUS E.TIME (GETP TABLE (QUOTE INTERVAL]
          [COND
            ((AND ID (OR (NOT (NUMBERP E.TIME))
                         (LESSP INIT E.TIME)))
              (PUT.INSTANCE TABLE ENTRY.TYPE ID (LIST NAME INIT E.TIME]
```

```
                (COND
                  [(GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE
                                                 (QUOTE CLASS-LABEL]
                  (T (SETQ CLASS NIL)))
                [COND
                  (WATCH.FN (SETQ ARG.LIST.1 (LIST TABLE NAME INIT E.TIME
                                                   (QUOTE DUMMY)
                                                   (CAR WATCH.FN)
                                                   NIL NIL]
                [COND
                  (SET.FN (SETQ ARG.LIST.2 (LIST TABLE NAME E.TIME E.TIME
                                                 (QUOTE NLESSP)
                                                 (CAR SET.FN)
                                                 (QUOTE LEVEL)
                                                 LEVEL]
                (COND
                  ((AND WATCH.FN SET.FN)
                    (SET.EL.DEMON TABLE NAME INIT E.TIME (QUOTE DUMMY)
                                  (CAR WATCH.FN)
                                  NIL NIL
                                  (LIST ARG.LIST.1 T ARG.LIST.2 ENTRY.TYPE
                                        ID INIT END CLASS))
                    (SET.EL.DEMON TABLE NAME E.TIME E.TIME (QUOTE NLESSP)
                                  (CAR SET.FN)
                                  (QUOTE LEVEL)
                                  LEVEL
                                  (LIST ARG.LIST.1 T ARG.LIST.2 ENTRY.TYPE
                                        ID INIT END CLASS)))
                  (WATCH.FN (SET.EL.DEMON TABLE NAME INIT END (QUOTE DUMMY)
                                          (CAR WATCH.FN)
                                          NIL NIL
                                          (LIST ARG.LIST.1 NIL ENTRY.TYPE ID
                                                INIT END CLASS)))
                  (SET.FN (SET.EL.DEMON TABLE NAME E.TIME E.TIME
                                        (QUOTE NLESSP)
                                        (CAR SET.FN)
                                        (QUOTE LEVEL)
                                        LEVEL
                                        (LIST ARG.LIST.2 NIL ENTRY.TYPE ID
                                              INIT END CLASS])

(ENTER.TYPE.3
  [LAMBDA (TABLE NAME INIT DURATION ENTRY.LIST ENTRY.TYPE ID CLASS)
    (PROG (A.LIST LST.1 LST.2 ARG.LIST.1 ARG.LIST.2 WATCH.FN SET.FN)
          (SETQ LST.1 (CAR ENTRY.LIST))
          (SETQ LST.2 (CADR ENTRY.LIST))
          (SETQ A.LIST (MAKE.ALIST.1 TABLE NAME LST.1 LST.2 ENTRY.TYPE
                                     ID CLASS))
```

```
[MAPC LST.1 (FUNCTION (LAMBDA (X)
        (TABLE.SETA TABLE NAME (CAR X)
                    (CDR X)
                    A.LIST]
[COND
  (ID (PUT.INSTANCE TABLE ENTRY.TYPE ID (LIST NAME LST.1
                                              LST.2]
[COND
  ([AND LST.1 (SETQ WATCH.FN
                    (GET.R.MODEL.PROP TABLE
                                      (LIST ENTRY.TYPE
                                            (QUOTE E-LIST)
                                            (QUOTE
                                              WATCH-DEMON]
    (SETQ ARG.LIST.1 (LIST TABLE NAME LST.1 (QUOTE DUMMY)
                           (CAR WATCH.FN)
                           NIL NIL]
[COND
  ([AND LST.2 (SETQ SET.FN
                    (GET.R.MODEL.PROP TABLE
                                      (LIST ENTRY.TYPE
                                            (QUOTE E-LIST)
                                            (QUOTE SET-DEMON]
    (SETQ ARG.LIST.2 (LIST TABLE NAME LST.2 (QUOTE NLESSP)
                           (CAR SET.FN)
                           (QUOTE LEVEL)
                           LEVEL]
(COND
  [(AND ARG.LIST.1 ARG.LIST.2)
    [MAPC LST.1
          (FUNCTION (LAMBDA (X)
              (SET.EL.DEMON TABLE NAME (CAR X)
                            (CDR X)
                            (QUOTE DUMMY)
                            (CAR WATCH.FN)
                            NIL NIL
                            (LIST ARG.LIST  RG.LIST.2
                                  ENTRY.T  INIT DURATION
                                  CLASS]
    (MAPC LST.2
          (FUNCTION (LAMBDA (X)
              (SET.EL.DEMON TABLE NAME (CAR X)
                            (CDR X)
                            (QUOTE NLESSP)
                            (CAR SET.FN)
                            (QUOTE LEVEL)
                            LEVEL
                            (LIST ARG.LIST.1 T ARG.LIST.2
                                  ENTRY.TYPE ID INIT DURATION
                                  CLASS]
```

```
        [ARG.LIST.1 (MAPC LST.1
                          (FUNCTION (LAMBDA (X)
                              (SET.EL.DEMON TABLE NAME (CAR X)
                                            (CDR X)
                                            (QUOTE DUMMY)
                                            (CAR WATCH.FN)
                                            NIL NIL
                                            (LIST ARG.LIST.1 NIL
                                                  ENTRY.TYPE ID INIT
                                                  DURATION CLASS]
        (ARG.LIST.2 (MAPC LST.2
                          (FUNCTION (LAMBDA (X)
                              (SET.EL.DEMON TABLE NAME (CAR X)
                                            (CDR X)
                                            (QUOTE NLESSP)
                                            (CAR SET.FN)
                                            (QUOTE LEVEL)
                                            LEVEL
                                            (LIST ARG.LIST.2 NIL
                                                  ENTRY.TYPE ID INIT
                                                  DURATION CLASS])


(MAKE.ALIST
  [LAMBDA (TABLE NAME INIT END ENTRY.TYPE ID CLASS)
    (PROG (MODEL A.LIST ID.LABEL CLASS.LABEL)
          (SETQ MODEL (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE)))
          (SETQ A.LIST (LIST (CONS (QUOTE ENTRY)
                                   ENTRY.TYPE)
                             (CONS (QUOTE STATE-NAME)
                                   (A.GETP MODEL (QUOTE STATE-NAME)))
                             (CONS (QUOTE LEVEL)
                                   (A.GETP MODEL (QUOTE LEVEL)))
                             (CONS (QUOTE START)
                             (CONS (QUOTE END)
                                   END)))
          (COND
            ((SETQ ID.LABEL (A.GETP MODEL (QUOTE ID-LABEL)))
              [COND
                ((NULL ID)
                  (PRIN1 "What ID?  ")
                  (SETQ ID (READ]
              (A.PUT A.LIST ID.LABEL ID)))
          (COND
            ((SETQ CLASS.LABEL (A.GETP MODEL (QUOTE CLASS-LABEL)))
              [COND
                ((NULL CLASS)
                  (MAPPRINQ ("What is " CLASS.LABEL "?  "))
                  (SETQ CLASS (READ]
              (A.PUT A.LIST CLASS.LABEL CLASS)))
          (RETURN A.LIST])
```

```
(MAKE.ALIST.1
  [LAMBDA (TABLE NAME LST.1 LST.2 ENTRY.TYPE ID CLASS)
    (PROG (MODEL A.LIST ID.LABEL CLASS.LABEL)
          (SETQ MODEL (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE)))
          (SETQ A.LIST (LIST (CONS (QUOTE ENTRY)
                                   ENTRY.TYPE)
                             (CONS (QUOTE STATE-NAME)
                                   (A.GETP MODEL (QUOTE STATE-NAME)))
                             (CONS (QUOTE LEVEL)
                                   (A.GETP MODEL (QUOTE LEVEL)))
                             (CONS (QUOTE ENTRIES)
                                   LST.1)
                             (CONS (QUOTE GAPS)
                                   LST.2)))
          (COND
            ((SETQ ID.LABEL (A.GETP MODEL (QUOTE ID-LABEL)))
              [COND
                ((NULL ID)
                  (PRIN1 "What ID?  ")
                  (SETQ ID (READ]
              (A.PUT A.LIST ID.LABEL ID)))
          (COND
            ((SETQ CLASS.LABEL (A.GETP MODEL (QUOTE CLASS-LABEL)))
              [COND
                ((NULL CLASS)
                  (MAPPRINQ ("What is " CLASS.LABEL "?  "))
                  (SETQ CLASS (READ]
              (A.PUT A.LIST CLASS.LABEL CLASS)))
          (RETURN A.LIST])

(PUT.INSTANCE
  [LAMBDA (TABLE ENTRY.TYPE ID LST)
    (PROG (TYPE.INST)
          (COND
            ((SETQ TYPE.INST (A.GETP# (GETP TABLE (QUOTE A.LIST))
                                      (LIST (QUOTE INSTANCES)
                                            ENTRY.TYPE)))
              (A.PUT TYPE.INST ID LST))
            (T (A.ADDPROP (GETP TABLE (QUOTE A.LIST))
                          (QUOTE INSTANCES)
                          (LIST ENTRY.TYPE (QUOTE DEFAULT)
                                (CONS ID LST])
```

B. Data Cancellation

Since much of the data is being entered with an ID, and since a call on a scheduler by a planner during replanning is likely to be through the ID of the plan, it is important to be able to cancel an assignment that is so designated. This is done by DELETE.ENTRY.ID or DELETE.ENTRY, depending on whether or not the entry has an ID and the ID is specified.

DELETE.ENTRY first looks for an occurence of the entry type, using FIND.ENTRY. If it can recover an ID from this entry, it calls DELETE.ENTRY.ID. Otherwise, it goes ahead on its own. The actual deletion of the entry in either function is done with CANCEL.ENTRY or, if type 3 and DELETE.ENTRY.ID, CANCEL.ENTRY.LST, given shortly. If there is an ID, DELETE.ENTRY.ID removes the entry from the instances in the table a-list. If the entry has a post-entry associated with it, each function calls itself to delete the post-entry.

Note that it would make little sense, generally, to have a type 3 entry without an ID. For a type 3 entry can be discontinuous. Without an ID, the seperate segments are not linked together except through the watch and set demons, if any. The deletion of a single segment will have a generally undesired side-effect of removing the watch and set demons from all segments, thus upsetting the procedure. Therefore, we assume that any such entry does have an ID, obtained if necessary by a GENSYM operation within the scheduler. This possibility is one reason why DELETE.ENTRY reverts to DELETE.ENTRY.ID if it can. It is also the reason why DELETE.ENTRY does not consider the possibility that the entry is type 3 after it has failed to revert to DELETE.ENTRY.ID.

DELETE.ENTRY.ID and DELETE.ENTRY use CANCEL.ENTRY and CANCEL.ENTRY.LST. These functions first remove the watch and set demons, as required, so that the demons will not be fired by the deletion. Then they call TABLE.SETA.TEST, given in section Vb6. TABLE.SETA.TEST is used rather than TABLE.SETA because of the intention to permit the use of DELETE.ENTRY, or DELETE.ENTRY.ID, by the watch and set demons. These demons may be fired as a result of changing the data in the table. We do not want the deletion to affect the data that may have already been changed. Hence TABLE.SETA.TEST is used since it tests the existing data before it makes any change.

The function, KILL.DEMON.PR, searches for an occurence of either the watch or set demon that may be specified by the model, and uses KILL.DEMON.FN to remove them.

These functions are defined as follows:

```
(DELETE.ENTRY.ID
  [LAMBDA (TABLE ENTRY.TYPE ID PRINT.SUP.FLG)
    (PROG (MODEL LST POST.ENTRY)
          (COND
            [(SETQ MODEL (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE]
            (PRINT.SUP.FLG (RETURN NIL))
            (T (MAPPRINQ ("No model named " ENTRY.TYPE " in " TABLE "."
                                            TERPRI))
               (RETURN NIL)))
          (COND
            [(SETQ LST (A.GETP# (GETP TABLE (QUOTE A.LIST))
                                (LIST (QUOTE INSTANCES)
                                      ENTRY.TYPE ID]
            (PRINT.SUP.FLG (RETURN NIL))
            (T (MAPPRINQ ("No entry of type " ENTRY.TYPE " with ID "
                                             ID " in " TABLE "."
                                             TERPRI))
               (RETURN NIL)))
          (COND
            ([EQ 3 (A.GETP# MODEL (QUOTE (E-LIST TYPE]
              (CANCEL.ENTRY.LST TABLE (CAR LST)
                                (2ND LST)
                                (3RD LST)
                                MODEL))
            (T (CANCEL.ENTRY TABLE (CAR LST)
                             (2ND LST)
                             (3RD LST)
                             MODEL)))
          (REM.INSTANCE TABLE ENTRY.TYPE ID)
          (COND
            ((AND (SETQ POST.ENTRY (A.GETP MODEL (QUOTE POST-ENTRY)))
                  (NUMBERP (3RD LST)))
              (DELETE.ENTRY.ID TABLE POST.ENTRY ID T)))
          (COND
            (PRINT.SUP.FLG (RETURN T))
            (T (PRIN1 "OK")
               (RETURN (CHARACTER 127])

(DELETE.ENTRY
  [LAMBDA (TABLE NAME ENTRY.TYPE TRIAL.TIME PRINT.SUP.FLG)
    (PROG (MODEL ENTRY POST.ENTRY)
          (COND
            [(SETQ MODEL (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE]
            (PRINT.SUP.FLG (RETURN NIL))
            (T (MAPPRINQ ("No model named " ENTRY.TYPE " in " TABLE "."
                                            TERPRI))
               (RETURN NIL)))
```

```
                [COND
                  [(NULL TRIAL.TIME (SETQ TRIAL.TIME (GETP TABLE (QUOTE START]
                  ((NOT (NUMBERP TRIAL.TIME))
                    (COND
                      (PRINT.SUP.FLG (RETURN NIL))
                      (T (PRIN1 "Can't.  TRIAL.TIME not a number.")
                         (RETURN (CHARACTER 127]
                [COND
                  [(SETQ TRIAL.TIME (FIND.ENTRY TABLE NAME TRIAL.TIME
                                                (A.GETP MODEL
                                                        (QUOTE STATE-NAME]
                  (PRINT.SUP.FLG (RETURN NIL))
                  (T (MAPPRINQ ("No occurence of " ENTRY.TYPE " in row "
                                                   NAME " in " TABLE "."
                                                   TERPRI))
                                                   TABLE "." TERPRI))
                     (RETURN (CHARACTER 127]
                (SETQ ENTRY (ST.ELT TABLE NAME TRIAL.TIME))
                (COND
                  ((AND (SETQ ID.LABEL (A.GETP MODEL (QUOTE ID-LABEL)))
                        (SETQ ID (A.GETP ENTRY ID.LABEL)))
                    (DELETE.ENTRY.ID TABLE ENTRY.TYPE ID T))
                  (T (CANCEL.ENTRY TABLE NAME (A.GETP ENTRY (QUOTE START))
                                   (A.GETP ENTRY (QUOTE END))
                                   MODEL)))
                (COND
                  ((SETQ POST.ENTRY (A.GETP MODEL (QUOTE POST-ENTRY)))
                    (DELETE.ENTRY TABLE NAME POST.ENTRY (A.GETP ENTRY
                                                                (QUOTE END))
                                  PRINT.SUP.FLG)))
                (COND
                  (PRINT.SUP.FLG (RETURN T))
                  (T (PRIN1 "OK")
                     (RETURN (CHARACTER 127])

(CANCEL.ENTRY
  [LAMBDA (TABLE NAME I.TIME E.TIME MODEL)
    (PROG (A.LIST LST WATCH.DEMON SET.DEMON TYPE)
          [SETQ TYPE (A.GETP# MODEL (LIST (QUOTE E-LIST)
                                          (QUOTE TYPE]
          (COND
            ((SETQ I.TIME (CHECK.TIME I.TIME E.TIME)))
            (T (RETURN NIL)))
          (KILL.DEMON.PR TABLE NAME I.TIME E.TIME MODEL TYPE)
          (TABLE.SETA.TEST TABLE NAME I.TIME E.TIME
                           (A.GETP MODEL (QUOTE STATE-NAME))
                           (GETP TABLE (QUOTE DEFAULT.LIST])
```

```
(CANCEL.ENTRY.LST
  [LAMBDA (TABLE NAME LST.1 LST.2 MODEL)
    (PROG (I.TIME E.TIME WATCH.DEMON SET.DEMON)
          (COND
            ((SETQ I.TIME (CHECK.TIME.LST LST.1)))
            (T (RETURN NIL)))
          [COND
            (LST.2 (SETQ E.TIME (CHECK.TIME.LST LST.2]
          (KILL.DEMON.PR TABLE NAME I.TIME E.TIME MODEL 3)
          (MAPC (CHECK.LST LST.1)
                (FUNCTION (LAMBDA (X)
                    (TABLE.SETA.TEST TABLE NAME (CAR X)
                                     (CDR X)
                                     (A.GETP MODEL (QUOTE STATE-NAME))
                                     (GETP TABLE (QUOTE DEFAULT.LIST])

(KILL.DEMON.PR
  [LAMBDA (TABLE NAME INIT END MODEL TYPE)
    (PROG (DEMON.FN TRIAL.TIME)
          (SETQ TRIAL.TIME (CHECK.TIME INIT END))
          (COND
            ((NULL TRIAL.TIME))
            ([SETQ DEMON.FN (CAR (A.GETP# MODEL
                                          (QUOTE (E-LIST WATCH-DEMON]
              (RETURN (KILL.DEMON.FN TABLE NAME TRIAL.TIME DEMON.FN
                                     TYPE)))
            ((NOT (NUMBERP END)))
            ([SETQ DEMON.FN (CAR (A.GETP# MODEL
                                          (QUOTE (E-LIST SET-DEMON]
              (RETURN (KILL.DEMON.FN TABLE NAME (NEXTHIGHER TABLE END)
                                     DEMON.FN TYPE])
```

XII DEMON AND DIALOG FUNCTIONS

There remain the functions used for the watch and set demons in the table. Other demons, used for alert requirements and to initiate more general system actions, must be designed to meet their specific purposes since we cannot define what they should do in general terms. The watch and set demons, however, which act to preserve the self-consistency of the scheduler or to maintain retroactive consist-y, ency are reasonably general. The basic requirement for them is that, when fired, they should reexecute the original entry if possible, or else cancel the entry.

The functions that have been defined for the watch and set demons are WATCH.1, WATCH.2 and WATCH.3, dpending on the type. The same function can generally be used for both the watch and set demon, although with different preconditions, as discussed before.

With all three of these functions, the first action, after recovering various data from the argument list, is to kill all occurences of the demon and of its complementary demon if any.

WATCH.1 then cancels the entry and then calls the function specified by the model to determine what to do about it. For MISSION, for example, DIALOG.1 is specified. Note that a type 1 entry must be made in its entirety or not at all. Hence any interference which causes WATCH.1 to be called requires its cancellation, although it can be re-instated in another row of the table if one is available.

WATCH.2 and WATCH.3 can themselves reenter the entry in a way to account for the change in the table data that caused the demon to be fired. This is true whether it is the watch or set demon that was fired. The key instruction in these functions is, therefore, the call of ENTER.TYPE. Note that this call is with the OVERRIDE flag set. It is therefore not necessary to remove the prior entries of the given type, which might fire other demons leading to other, undesired, side-effects.

All three of these functions must account for the possibility that the system clock may have advanced considerably since the original entry. In fact, the start of the table may be after that of the of the original entry. Therefore, adjustments need to be made. In WATCH.3, two special functions are used for these adjustments. CORRECT.DURATION adjusts the duration to account for that part of the entry that has been completed. CORRECT.LIST takes a list of dotted pairs identifying intervals of time, and compares it with a specified time, removing any dotted pairs that are before the specified time

and correcting any pairs that start before the specified time. It is used to determine which, if any, of the original entries must be removed.

Note that WATCH.1, 2, and 3 remove entries, entering in their place the default A.list specified by the table, only as needed. This is of some importance since otherwise other demons might be fired accidentally, with possible undesired side-effects.

WATCH.1 cancels out the entry. In the case of MISSION, as its model is specified in Table 4, DIALOG.1 is then called, asking the user if the assignment should be reinstated with another pilot. Alternatively, a function could be called to make the selection of a substitute pilot, according to some specified criterion. The change can be made by modifying the model for MISSION in the scheduler, specifying a different CALL-FUNCTION.

Table 9 gives a short exercise that illustrates the operation of some of these functions. The initial state of the scroll table is shown first; e.g., ABLE is sick indefinitely, and BAKER has been assigned to a mission. Data is then entered that BAKER is sick from the present time to 8:30. The demon is fired by this entry. WATCH.1 first cancels out this assignment, including the rest period following it. It then calls DIALOG.1 which issues an alert message and reports that CHARLES, DAVIS and ELLIS are available. The user chooses DAVIS, which becomes the value that DIALOG.1 returns. WATCH.1 then reinstates the assignment with DAVIS as the pilot.

Table 9

A SAMPLE SCENARIO ILLUSTRATING
DEMON BEHAVIOR

9A
INITIAL CONDITION

```
(PRINT.PROP 'PILOTS]

   Table:  PILOTS
Property:  STATE-NAME          (AVAIL replaced by *.)

Name\Time      00:00      06:30      12:00      18:00
-------------------------------------------------------
ABLE       |   SICK       SICK       SICK       SICK
BAKER      |   *          ASG        ASG.RET    *
CHARLES    |   *          *          *          *
DAVIS      |   *          *          *          *
ELLIS      |   *          *          *          *
```

9B

DATA ENTRY

```
(ENTER.TYPE 'PILOTS 'BAKER 0 500 'SICK NIL 'S2]

                **ALERT**

                *CONFLICT*

MISSION with ID M1 in conflict with entry of state SICK.
The available rows are:  CHARLES, DAVIS, ELLIS.
Which row should be used?  (Name or ] to cancel.)  DAVIS
OK
```

9C

FINAL CONDITON

```
(PRINT.PROP 'PILOTS]

   Table:  PILOTS
Property:  STATE-NAME          (AVAIL replaced by *.)

Name\Time      00:00      06:30      08:30      12:00      18:00
-----------------------------------------------------------------
ABLE       |   SICK       SICK       SICK       SICK       SICK
BAKER      |   SICK       SICK       *          *          *
CHARLES    |   *          *          *          *          *
DAVIS      |   *          ASG        ASG        ASG.RET    *
ELLIS      |   *          *          *          *          *
```

The functions described in this section are defined as follows:

```
(WATCH.1
  [LAMBDA (OLD.ALIST NEW.ALIST ARG.LIST)
    (PROG (TABLE NAME ENTRY.TYPE I.TIME E.TIME CLASS.LABEL CLASS
                 CALL.FN NEW.NAME)
          (SETQ TABLE (CAAR ARG.LIST))
          (SETQ NAME (2ND (CAR ARG.LIST)))
          (SETQ ENTRY.TYPE (3RD ARG.LIST))
          (SETQ I.TIME (A.GETP OLD.ALIST (QUOTE START)))
          [COND
            ((LESSP I.TIME (GETP TABLE (QUOTE START)))
              (SETQ I.TIME (GETP TABLE (QUOTE START]
```

```
            [COND
              [(SETQ E.TIME (A.GETP OLD.ALIST (QUOTE END)))
                (COND
                  ((LESSP E.TIME I.TIME)
                    (RETURN NIL]
              (T (SETQ E.TIME (QUOTE INDEF]
            [COND
              ([SETQ CLASS.LABEL (GET.R.MODEL.PROP TABLE
                                                   (LIST ENTRY.TYPE
                                                     (QUOTE CLASS-LABEL]
                                                     (QUOTE CALL-FUNCTION]
            (STRIP.ENTRY.1 TABLE NAME I.TIME E.TIME ENTRY.TYPE ARG.LIST)
            (COND
              ((SETQ ID (4TH ARG.LIST))
                (REM.INSTANCE TABLE ENTRY.TYPE ID)))
            [COND
              ([SETQ CALL.FN (GET.R.MODEL.PROP TABLE (LIST ENTRY.TYPE
                                                           (QUOTE E-LIST)
                                                           (QUOTE
                                                             WATCH-DEMON)
                                                           (QUOTE
                                                           CALL-FUNCTION]
                (SETQ NEW.NAME (APPLY* CALL.FN TABLE NAME OLD.ALIST
                                       NEW.ALIST ENTRY.TYPE ID)))
              (T (MAPPRINQ (ENTRY.TYPE " for " NAME " in " TABLE
                                       " cancelled because of conflict."
                                       TERPRI))
                 [COND
                   (ID (MAPPRINQ ("ID of entry was " ID
                                                     ".  Entry was from "
                                                     I.TIME " to " E.TIME
                                                     ".")))
                   (T (MAPPRINQ ("Entry was from " I.TIME " to " E.TIME
                                                   "."]
                 (RETURN (CHARACTER 127]
            (COND
              (NEW.NAME (ENTER.TYPE TABLE NEW.NAME I.TIME E.TIME
                                    ENTRY.TYPE ID CLASS T))
              (T (PRIN1 "Entry cancelled.")))
            (RETURN (CHARACTER 127])

(WATCH.2
  [LAMBDA (OLD.ALIST NEW.ALIST ARG.LIST)
    (PROG (TABLE NAME INIT LST ENTRY.TYPE ID END TIME.2 CLASS E.TIME)
          (SETQ TABLE (CAAR ARG.LIST))
          (SETQ NAME (2ND (CAR ARG.LIST)))
          (SETQ LST (AUTO.KILL.DEMON ARG.LIST))
          (SETQ ENTRY.TYPE (CAR LST))
```

```
                (COND
                  ((SETQ ID (2ND LST))
                    (REM.INSTANCE TABLE ENTRY.TYPE ID)))
                (SETQ INIT (3RD LST))
                (SETQ END (4TH LST))
                (SETQ CLASS (5TH LST))
                [COND
                  ((LESSP INIT (GETP TABLE (QUOTE START)))
                    (SETQ INIT (GETP TABLE (QUOTE START]
                (COND
                  ((NOT (NUMBERP END))
                    (SETQ TIME.2 NIL))
                  ((LESSP INIT END)
                    (SETQ TIME.2 (IDIFFERENCE END INIT)))
                  (T (RETURN NIL)))
                (SETQ E.TIME (CHECK.ENTRY.2 TABLE NAME INIT END
                                            (GET.R.MODEL.PROP TABLE
                                                              (LIST ENTRY.TYPE
                                                               (QUOTE LEVEL)))
                                            T))
                [COND
                  ((AND (NUMBERP E.TIME)
                        (LESSP E.TIME END))
                    (TABLE.SETA.TEST TABLE NAME E.TIME END
                                     (GET.R.MODEL.PROP TABLE
                                                       (LIST ENTRY.TYPE
                                                         (QUOTE STATE-NAME)))
                                     (GETP TABLE (QUOTE DEFAULT.LIST]
                (ENTER.TYPE TABLE NAME INIT TIME.2 ENTRY.TYPE ID CLASS T T])

(WATCH.3
  [LAMBDA (OLD.ALIST NEW.ALIST ARG.LIST)
    (PROG (TABLE NAME LST ENTRY.TYPE ID INIT DURATION CLASS OLD.LAST
                 NEW.LAST)
                (SETQ TABLE (CAAR ARG.LIST))
                (SETQ NAME (2ND (CAR ARG.LIST)))
                (SETQ LST (AUTO.KILL.DEMON.1 ARG.LIST))
                (SETQ ENTRY.TYPE (CAR LST))
                (COND
                  ((SETQ ID (2ND LST))
                    (REM.INSTANCE TABLE ENTRY.TYPE ID)))
                (SETQ INIT (3RD LST))
                (SETQ DURATION (4TH LST))
                (SETQ CLASS (5TH LST))
                [SETQ DURATION (CORRECT.DURATION (GETP TABLE (QUOTE START))
                                                 DURATION
                                                 (3RD (CAR ARG.LIST]
                [COND
                  ((LESSP INIT (GETP TABLE (QUOTE START)))
                    (SETQ INIT (GETP TABLE (QUOTE START]
```

```
        (ENTER.TYPE TABLE NAME INIT DURATION ENTRY.TYPE ID CLASS T T)
        [SETQ OLD.LAST (LAST.ENTRY (3RD (CAR ARG.LIST]
        (SETQ NEW.LAST (LAST.ENTRY (CHECK.ENTRY.3 TABLE NAME INIT
                                                  DURATION
                                                  (GET.R.MODEL.PROP
                                                    TABLE
                                                    (LIST ENTRY.TYPE
                                                      (QUOTE LEVEL)))
                                                  T)))
        (COND
          ((NOT (NUMBERP NEW.LAST)))
          ((OR (NOT (NUMBERP OLD.LAST))
               (LESSP NEW.LAST OLD.LAST))
            (TABLE.SETA.TEST.LST TABLE NAME (CORRECT.LIST
                                      (NEXTHIGHER TABLE NEW.LAST)
                                      (3RD (CAR ARG.LIST)))
                                 (GET.R.MODEL.PROP TABLE
                                                   (LIST ENTRY.TYPE
                                                         (QUOTE
                                                         STATE-NAME)))
                                 (GETP TABLE (QUOTE DEFAULT.LIST])

(CORRECT.DURATION
  [LAMBDA (START DURATION LST.1)
    (PROG (NIL)
          [MAPC LST.1 (FUNCTION (LAMBDA (X)
                    (COND
                      ((LESSP (CAR X)
                              START)
                        (COND
                          [(OR (NOT (NUMBERP (CDR X)))
                               (GREATERP (CDR X)
                                         START))
                            (SETQ DURATION (IPLUS DURATION (CAR X)
                                                  (IMINUS START]
                          (T (SETQ DURATION (IPLUS DURATION (CAR X)
                                                   (IMINUS (CDR X]
          (RETURN DURATION])
```

```
(CORRECT.LIST
  [LAMBDA (TIME LST)
    (PROG (NEW.LIST)
          [MAPC LST (FUNCTION (LAMBDA (X)
                    (COND
                      ((OR (GREATERP (CAR X)
                                     TIME)
                           (EQP (CAR X)
                                TIME))
                        (SETQ NEW.LIST (CONS X NEW.LIST)))
                      ((NOT (NUMBERP (CDR X)))
                        (SETQ NEW.LIST (CONS (CONS TIME (CDR X))
                                             NEW.LIST)))
                      ((GREATERP (CDR X)
                                 TIME)
                        (SETQ NEW.LIST (CONS (CONS TIME (CDR X))
                                             NEW.LIST]
          (RETURN NEW.LIST])

(DIALOG.1
  [LAMBDA (TABLE NAME OLD.ALIST NEW.ALIST ENTRY.TYPE ID)
    (PROG (NAMELIST X)
          [MAPPRINQ ((TAB 20)
                     "**ALERT**"
                     (RPTQ 2 (TERPRI))
                     (TAB 20)
                     "*CONFLICT*"
                     (RPTQ 2 (TERPRI]
          [COND
            (ID (MAPPRINQ (ENTRY.TYPE " with ID " ID
                                      " in conflict with entry of state "
                                       (A.GETP NEW.ALIST
                                               (QUOTE STATE-NAME))
                                       "." TERPRI)))
            (T (MAPPRINQ (ENTRY.TYPE " for " NAME " from "
                                     (A.GETP OLD.ALIST (QUOTE START))
                                     " to "
                                     (A.GETP OLD.ALIST (QUOTE END))
                                     " in conflict with entry of state "
                                     (A.GETP NEW.ALIST
                                             (QUOTE STATE-NAME))
                                     "." TERPRI]
          (COND
            [(SETQ NAMELIST (CHECK.NAMES.1 TABLE (A.GETP OLD.ALIST
                                                         (QUOTE START))
                                           (A.GETP OLD.ALIST (QUOTE END))
                                           (A.GETP OLD.ALIST (QUOTE LEVEL]
            (T (PRIN1 "No alternatives available.")
               (TERPRI)
               (RETURN NIL)))
          (MAPRINT NAMELIST T "The available rows are:  " "." ", ")
```

XIII CONCLUSIONS

We have described the design of a scheduler in ACS.1 in considerable detail, defining the relevent functions that are used in its basic implementation.

In this discussion, we have limited ourselves largely to those features, and their implementations, that provide the basic capabilities required of a scheduler. Other features can be added for convenience or to meet special requirements. Some of these features have, in fact, been implemented, although they have not been included in this report. The functions given provide a foundation on which additional capabilities can be built as required. The functions described implement a number of interesting technical features. Among these are the following:

The use of a scroll table as the structure within which to retain the information given to the system, or developed by it, that may affect the future availability of the designated type of resource.

The encoding of the models that collectively form a resource model as a data structure that is entirely separate from the functions that manipulate the data. The models, therefore, are available for modification or extension, and such changes can be made without requiring the manipulative functions to be reprogrammed; and with only minimal knowledge of the scheduler's operations.

The recognition of the property that we call "retroactive consistency" as that required to maintain the self-consistency of the data held by the scheduler.

The use of demons to enforce retroactive consistency.

The specification of the format of a demon so as to allow a simple implementation of the self-destruct property of a demon. The self-destruct property is recognized as being of major importance in the application environment being studied. It has been used previously, but not identified as such. Its explicit recognition has lead to a possibly unique formulation of the demon structure.

While there are a large number of ways in which this work can be extended, two extensions seem to be of particular importance:

Context Capability. The ability to hold and use data that is true only in some possible contingency. Conflicting data can then be admitted, providing the alternatives can be identified as applying under different conditions. Contingency plans are an example. The work described here has already been extended to include the basic capabilities required for this type of operation, but it has not yet been integrated into the complete design of a scheduler.

Continuous Operations. The ability to handle conditions where the desired state requires a continuing condition. The continuity of the condition may require an indefinite sequence of assignments to different resources of a given type. This may impose a need for relationships that link entries in different rows of the scroll table at different times. There is no apparent reason why this should not be possible, but the means for accomplishing it remain to be developed. No effort has been made in this direction as yet, but it is recognized as desirable to increase the scope of the system concept used in ACS.1.